Nuno Daniel Gouveia de Sousa Grade

Licenciado em Engenharia Informática

Data queries over heterogeneous sources

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadores : João Costa Seco, Professor Doutor,
Universidade Nova de Lisboa
Lúcio Ferrão, Chief Architect,
OutSystems

Júri:

Presidente:

Arguente:

Vogal: Prof. Dr. João Costa Seco

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2013

Data queries over heterogeneous sources

Acknowledgements

I would like to show my gratitude to the orientation received from both of my supervisors João Seco and Lúcio Ferrao, mainly from the orientation received at *OutSystems* that allowed me to improve my research and solution architecture and development skills, after this dissertation project. Besides, they both taught me how to improve my English writing skills for scientific documents like this. I also thank Nuno Preguiça, and mostly José Alferes, both professors of our university, for being so available and helpful, giving good advices and feedback when help was requested.

Further, I also express my gratitude for the collaboration between the university and *OutSystems*, which allowed me to receive a scholarship during the development of this project, to the three colleagues André Simões, Miguel Alves, and Tiago Almeida, developing their thesis at *OutSystems* as well, and to Sérgio Silva. They played an important role because we constantly discussed the solutions we were creating and the material we were finding to each others, therefore receiving continuous feedback that contributed to the success of the project.

Finally, to my family, closer friends, and specially my girlfriend, who also played an important role due to the motivation they gave me during the development of this dissertation, mostly in the most difficult times of the investigation phase, when no solution seemed to rise.

Abstract

Enterprises typically have their data spread over many software systems, such as custom made applications, CRM systems like SalesForce, CMS systems, or ERP systems like SAP. In these setting, it is often desired to integrate information from many data sources to accomplish some business goal in an application. Data may be stored locally or in the cloud in a wide variety of ways, demanding for explicit transformation processes to be defined, reason why it is hard for developers to integrate it. Moreover, the amount of external data can be large and the difference of efficiency between a smart and a naive way of retrieving and filtering data from different locations can be great. Hence, it is clear that developers would benefit greatly from language abstractions to help them build queries over heterogeneous data sources and from an optimization process that avoids large and unnecessary data transfers during the execution of queries.

This project was developed at *OutSystems* and aims at extending a real product, which makes it even more challenging. We followed a generic approach that can be implemented in any framework, not only focused on the product of *OutSystems*.

Keywords: Data integration, Web services, SalesForce, SAP, Query optimization, Remote data sources, LINQ, Statistics cache, Developer hints, Adaptive query execution

Resumo

É normal a informação utilizada nos sistemas de software empresariais estar espalhada em várias fontes, tais como aplicações desenvolvidas à medida, sistemas CRM como o SalesForce, sistemas CMS, ou sistemas ERP como o SAP. É portanto normal ser necessário integrar informação proveniente de várias fontes para atingir algum objectivo de negócio numa aplicação. A informação pode estar guardada em bases de dados locais, ou na cloud, de várias maneiras, requerendo assim a definição de processos explícitos de transformação, sendo por isso difícil para os programadores produzirem estas integrações. Além disso, a quantidade de dados externos pode ser grande e a diferença de eficácia entre uma abordagem inteligente ou ingénua de devolver e filtrar informação proveniente de várias fontes, pode ser enorme. Assim, é claro que os programadores beneficiariam bastante de linguagens de abstracção para os ajudar a construir queries sobre fontes de informação heterogéneas e de processos de optimização que evitem a transferência de grandes e desnecessárias quantidades de dados, durante a execução de queries.

Este projecto foi desenvolvido na *OutSystems* e tem como objectivo estender um producto real, o que o torna ainda mais desafiante. Nós seguimos uma abordagem genérica que pode ser implementada em qualquer sistema, não sendo portanto, focada somente no produto da *OutSystems*.

Palavras-chave: Integração de dados, Web services, SalesForce, SAP, Optimização de queries, Data sources remotos, Linq, Re-Linq, Cache de estatísticas, Hints de programadores, Execução adaptativa de queries

Contents

List of Figures

List of Tables

Listings

1

Introduction

Internet and its services have been growing exponentially over the last decades, establishing a strong impact on data repositories, data sharing and systems cooperation. Enterprise applications typically use data in their local databases and data arising from external services in order to combine them and achieve some goal. As data storages and processing hardware power increases, developers often deal with large amounts of data. Although these technological improvements have several benefits, they also come with a price. If we consider large amounts of data being transferred over the web from different kind of systems, using different formats of data can turn out to become a problem, and the transfer of large data sets slows down every process due to network transfer latency. To produce these integrations, systems must combine information from heterogeneous data sources. Developers have a hard time integrating external data sources, usually requiring hand-made custom data loading, filtering, processing and adaptation algorithms.

1.1 Motivation

Enterprise applications usually combine information from heterogeneous sources. Typically, there are multiple ways to fetch data from these data sources, for instance, SQL for writing queries against relational databases, and the programming language *C#* to invoke APIs of web services.

As a possible application scenario, we built a web application for managing lawyers, clients, courts, etc... Registered lawyers can check, edit, and create their processes and clients, as well as see a list of available courts. A database local to the application stores

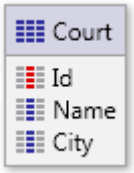


Figure 1.1: Database courts data

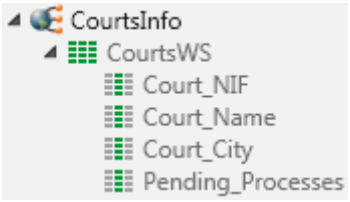
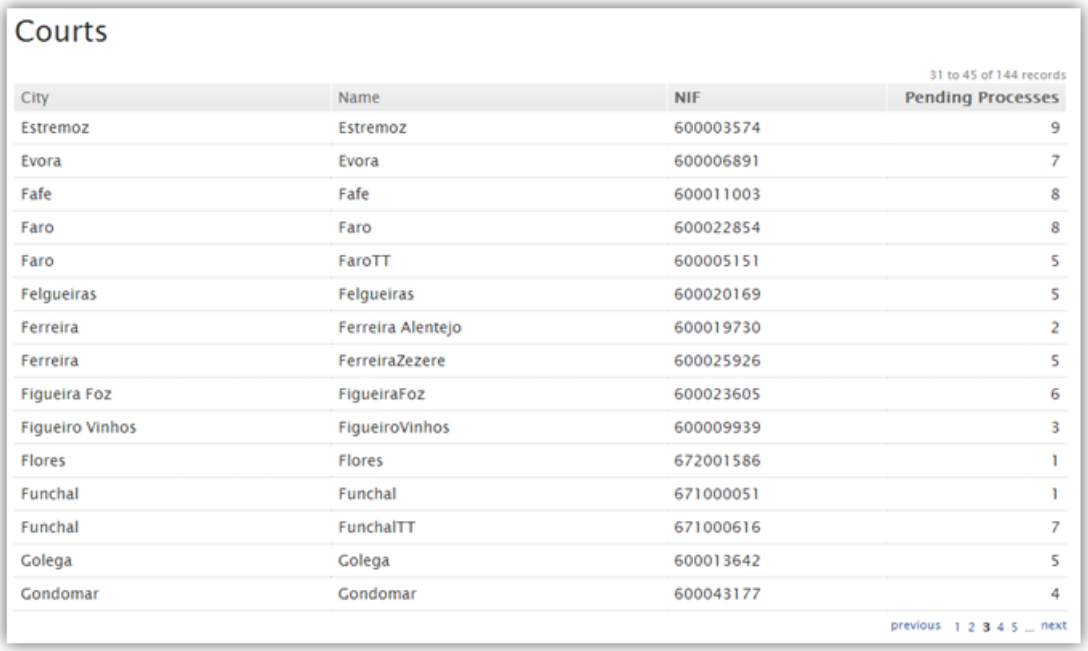


Figure 1.2: Web service courts data



Courts

31 to 45 of 144 records

City	Name	NIF	Pending Processes
Estremoz	Estremoz	600003574	9
Evora	Evora	600006891	7
Fafe	Fafe	600011003	8
Faro	Faro	600022854	8
Faro	FaroTT	600005151	5
Felgueiras	Felgueiras	600020169	5
Ferreira	Ferreira Alentejo	600019730	2
Ferreira	FerreiraZezere	600025926	5
Figueira Foz	FigueiraFoz	600023605	6
Figueiro Vinhos	FigueiroVinhos	600009939	3
Flores	Flores	672001586	1
Funchal	Funchal	671000051	1
Funchal	FunchalTT	671000616	7
Golega	Golega	600013642	5
Gondomar	Gondomar	600043177	4

previous 1 2 3 4 5 ... next

Figure 1.3: Integration result

information about courts. Financial details about courts (NIF[1], pending processes) are supplied via a web service. The structure of the data arising from these two data sources is shown in Figure 1.1 and 1.2.

In order to provide a list of available courts, data must be retrieved from a database and from a web service and then combined. The combined data can then be used to produce the application screen shown in Figure 1.3.

In order to combine data from these data sources, a developer must explicitly fetch the data from the database and from the web service, and then integrate and filter the desired results. However, if not implemented efficiently, such operations may be time-consuming.

Therefore, we aim at providing a simple and effective abstraction to express queries involving local and external data sources. The resulting code should be at least as efficient as the hand written one. The average programmer tends to reach rather naive and

[1]Tax Identification Number in Portuguese

inefficient solutions to this problem. A highly skilled developer reaches higher efficiency but usually resourcing to more complicated and harder to maintain code.

Hence, the motivation of this work is to look into real-world data integration scenarios, and design an efficient, reliable and user-friendly solution that could in principle be integrated in the *Agile Platform* (section 1.3.1).

1.2 Problems, Goals, and Challenges

A problem of developing an efficient solution for these integration scenarios concerns the regular changes of data within the data sources, requiring possible synchronization mechanisms. Instead, applications can access external data and further use it, saving only desired data. While accessing and querying external data is not a big issue, merging it with internal data and data arising from several other external sources, in an efficient way, is difficult and problematic. Developers may not be aware of the performance of their executions and as data evolves, the best execution plans change. As for the development complexity, there are many possible combinations to execute these integrations and the more efficient queries/APIs to write against the data sources may change for different queries.

In order to implement the integration scenario presented in Figure 1.3, some naive approaches may be developed. Some of these approaches are inefficient because they do not follow an efficient execution flow and besides they do not implement optimization techniques, such as the use of hashing techniques to compute the merge of collections in memory. The code snippets of Figures 1.4, 1.5 and 1.6 show three different C# naive implementations that produce the integration. In the first approach, the developer starts by fetching all the data from the web service and from the database, maintaining the data in separate data sets. Afterwards, it iterates both data sets and manually matches the courts by name, producing the desired integration. In the second approach, the developer fetches all the data from the web service and, for each record retrieved, it queries the database for specific courts, manually building a list with the merging result. Finally, in the last approach the developer gets all the data from the database and, for each record retrieved, it queries the web service for a specific court, manually building a list with the merging result.

We want to build queries using a common language, similar to SQL (section 1.3.2) for both databases and web services because SQL is world-wide used for querying relational databases and it is the query language used by *OutSystems*. Furthermore, we want to be able to join databases and web services seamlessly in a query and ensure an efficient execution.

However, there is a lack of common query languages for both databases and web services. Besides, as a system and its data evolves, the awareness about those changes is

```
// get all ws courts
External_Court[] ext_courts = courts_webservice.GetAll();

// get all db courts
IEnumerable<OSUSR_D95_COURT11> db_courts = new databaseDataContext().ExecuteQuery<OSUSR_D95_COURT11>
    ("SELECT * FROM OSUSR_D95_COURT11", new object[] { });

// compute join
foreach (External_Court ws_court in ext_courts) {
    foreach (OSUSR_D95_COURT11 db_court in db_courts) {
        if (ws_court.Court_Name.Equals(db_court.NAME)) {
            // build record and append to result list
            result.Add(new { City = db_court.CITY, Name = db_court.NAME, NIF = ws_court.Court_NIF });
        }
    }
}
```

Figure 1.4: First naive approach

```
// get all ws courts
External_Court[] ext_courts = courts_webservice.GetAll();

// compute join
foreach (External_Court ws_court in ext_courts) {

    // find related db court
    OSUSR_D95_COURT11 db_court = new databaseDataContext().ExecuteQuery<OSUSR_D95_COURT11>
        ("SELECT * FROM OSUSR_D95_COURT11 WHERE NAME = '" + ws_court.Court_Name + "'", new object[] { })
        .First();

    // build record and append to result list
    result.Add(new { City = db_court.CITY, Name = db_court.NAME, NIF = ws_court.Court_NIF });
}
```

Figure 1.5: Second naive approach

not rich and therefore developers do not have enough expected context.

Hence, we try to answer the following questions:

- Is it possible to get parts of the data from the external data sources by filtering it regarding to some criteria?
- Is it possible to express the merging of internal and external data using a single query?
- Can we avoid store external data in the database and still perform these operations efficiently?

1.3 Context

In this section we give a wider context about the company on which this work is being developed and describe several concepts further spoken in this document.

This work inserts itself in the context of a collaboration between the investigation group of programming languages of CITI (Centro de Informática e Tecnologias de Informação), of the Department of Informatics of FCT/UNL and the company *OutSystems*.

```
// get all db courts
IEnumerable<OSUSR_D95_COURT11> db_courts = new databaseDataContext().ExecuteQuery<OSUSR_D95_COURT11>
    ("SELECT * FROM OSUSR_D95_COURT11", new object[] { });

// compute join
foreach (OSUSR_D95_COURT11 db_court in db_courts) {

    // find related ws court
    External_Court ws_court = courts_webservice.GetByCourt_Name(db_court.NAME);

    // build record and append to result list
    result.Add(new { City = db_court.CITY, Name = db_court.NAME, NIF = ws_court.Court_NIF });
}
```

Figure 1.6: Third naive approach

The focus of this work resides over *OutSystems Agile Platform*, a platform that provides a DSL[2] to manipulate action flow behaviours.

This project is inspired by *OutSystems* use cases and it was developed inside its R&D department. Besides, this project is not focused nor directly connected to the *Agile Platform* because we propose a general solution that may be applied in any development environment. Nevertheless, this work has the mission of proposing to the platform an easier, semi-automatic and optimized process of transforming and integrating data from several data sources.

1.3.1 *OutSystems* DSL

OutSystems is a multinational software company founded in 2001 operating in the agile software development market. The mission of the company is to provide technology that speeds up and reduces the costs of the delivery and management of web business applications using agile methodologies.

The product of *OutSystems* is called *OutSystems Agile Platform*, a platform aiming the full life cycle of delivering and managing web business applications. It includes the tools required to integrate, develop, deploy, manage and change web business applications. The platform maintains a powerful graphic environment tool called ServiceStudio, where developers can design and develop web applications. It also allows to automate a variety of interactions and processes, so that no coding and hard work are necessary to implement several kind of features. For further details, check the *OutSystems* website[3].

OutSystems Agile Platform

OutSystems Agile Platform provides a rich development environment where a developer can build web applications. It also provides easy access and integration of external data sources like web services. The *Agile Platform* offers methods to create and maintain web business applications that can be in constantly change under an easy, fast and incremental

[2]Domain-Specific Language (DSL) is a programming language dedicated to a particular problem domain.
[3]http://www.outsystems.com

way. It is possible to obtain functional solutions ready for production within a short time and with little effort, as well as adding new features to an application when it is needed. Thus, it is simple to present provisional versions and change them according to user feedback over the time, increasing the life cycle of programs developed with this tool. The main goal of this approach is to speed-up the entry of products in the market and fill their needs with flexibility. Figure 1.7 illustrates part of the architecture of the platform.

Figure 1.7: Part of *Agile Platform* architecture

The part of *Agile Platform* which is related to our work is Service Studio, an IDE[4] that allows the easy creation of web applications able to integrate external components. This IDE provides a powerful graphical environment, allowing a developer to build web applications using visual models.

Service Studio

Service Studio offers to developers the ability to visually model the way web pages of a web application look like. Those pages often include components like tables, buttons, forms, among others. This tool allows to associate actions to manipulate these components, according to users interaction or at the page loading moment. In order to model all this behaviour, a simple visual programming language is provided.

Regarding to web services (section 2.1), it is also possible to integrate them with Service Studio by adding references to web services in the appropriate place in the project. Afterwards, a user is able to fetch and show the data arising from a web service, as well as iterating over the fetched structure to perform some transformation.

As for the data model, it can be graphically and easily changed in Service Studio. When changing the data model, the impact is propagated to the whole application in the way of warnings, shown in the graphical environment as well. Until a user solves those warnings the application is not be ready to be published. Figure 1.8 shows the data model screen part in Service Studio, where a user can perform changes graphically.

[4]Integrated development environment, a software application that provides comprehensive facilities to

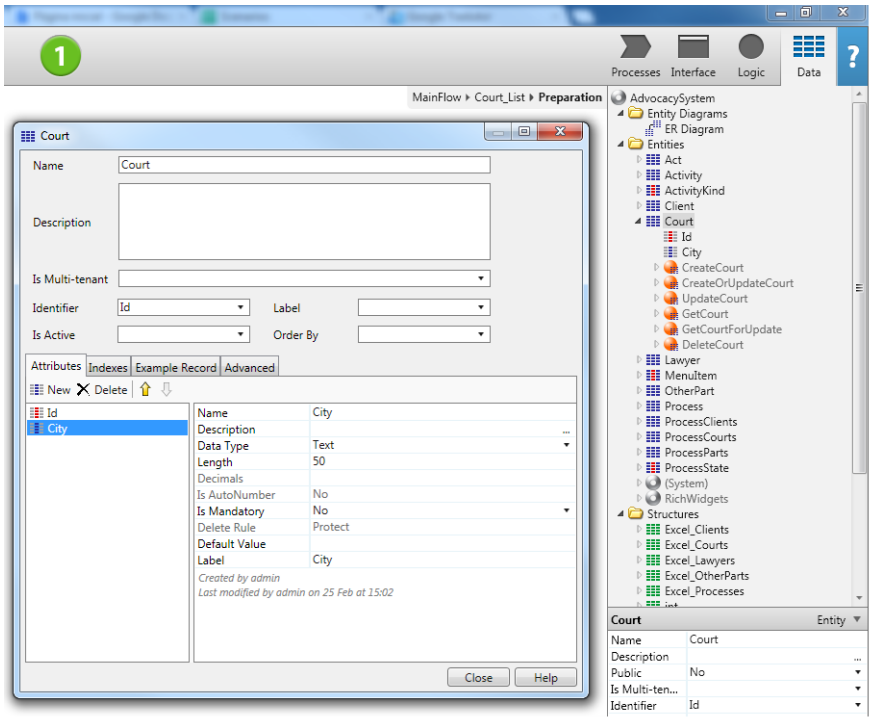


Figure 1.8: Interacting with the data model

Simple Queries and Advanced Queries

These two graphical abstractions provide to the user the capability of querying the database. In spite of having the same goal, they provide different features. Hence, in order to allow users who do not have knowledge about the database query language SQL (section 1.3.2), Simple Query is a graphical interface which eases the creation of queries allowing the specification of input parameters, entities, conditions and sorting. Figure 1.9 shows an example of a Simple Query retrieving all the courts from the database.

The limitations of this abstraction appear when a developer needs to write more complex queries, using aggregation functions or group by clauses (section 3.2.1), for instance. For the cases where the expressiveness of the Simple Query is not enough, the operation Advanced Query delegates to the developer the responsibility of creating the query on a language that is close to standard SQL. Besides, developers can use the Advanced Query operation to code SQL directly. To what concerns querying data from several data sources, it is not possible to query data from database tables and web services within a Simple or an Advanced Query.

1.3.2 Query languages

A query language is a programming language used to query information systems, such as databases. A query language can be classified according to whether it is querying

computer programmers for software development.

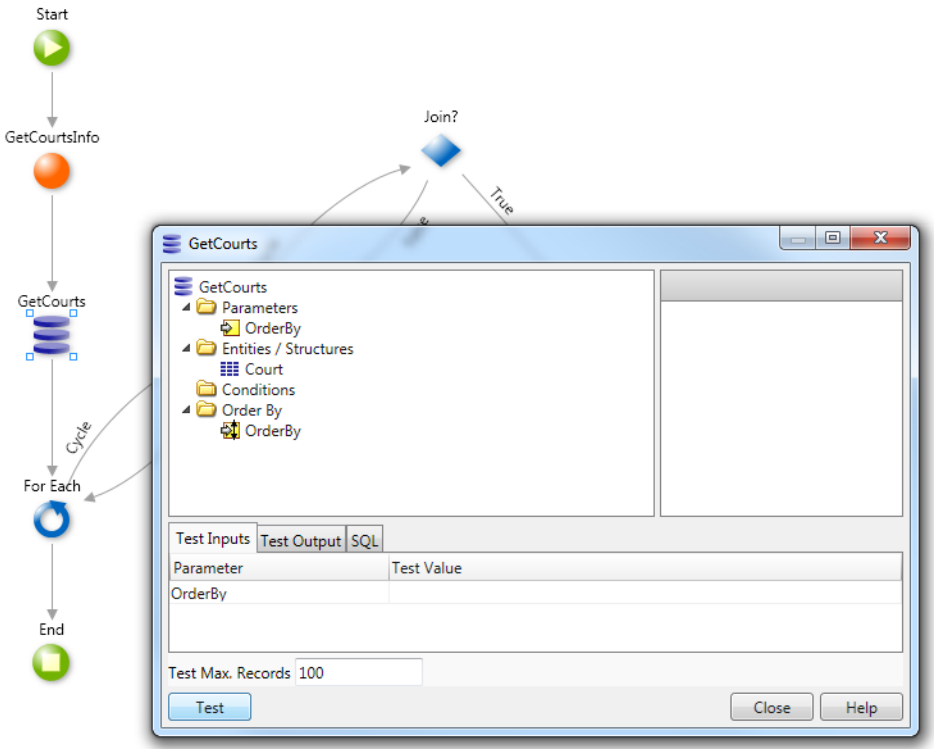


Figure 1.9: Simple Query

databases or information retrieval services. The difference resides in the fact that while a database query attempts to give factual answers to factual questions, an information retrieval query attempts to find documents containing information that is relevant to an area of inquiry.

A query is an expression that retrieves data from a data source and it is usually expressed in a specialized query language. To our context, we focus on structured queries similar to the ones performed over databases. The data sources focused in this project are databases and web services, which we treat as having similar structures. Thus, document retrieval is not included in our research.

SQL: A declarative programming language developed to search information in relational databases [SKS10, sita]. This language is the world most common language used to query databases due to its simplicity and ease of use. SQL allows to explicitly specify the result form of the query, although not the way to get it. Typically, the language can be migrated between platforms without structural changes. As for database systems that use SQL, the most famous and world-wide used are Oracle, MS SQL Server, MySQL, PostgreSQL and DB2.

1.3.3 Data transfer formats

XML: e**X**tensible **M**arkup **L**anguage [HM04] is one of the most used data exchange languages. It provides an information inter-change format that is editable, easily parsed, and

capable of representing nearly any kind of structured or semi-structured information. Although the design of XML resides on documents, it is broadly used for the representation of certain data structures, for instance in web services, where data is received and sent in XML.

As XML is a data representation format, query languages for XML documents also exist. **XPath** (XML Path Language [HM04]) is a query language for selecting nodes from an XML document. Besides, it also allows to compute values from the content of a document. XPath expressions identify one or more of internal components of an XML document and is widely used in programming libraries for accessing XML-encoded data. Moreover, another query language for XML is **XQuery** [Wal07]. XQuery is a query and functional programming language developed to query sets of XML data, which provides flexible facilities to extract information from real and virtual documents over the web. In addition, XQuery is a language for interaction between the web and databases, since the access pattern is the same.

1.4 Methodologies and approaches

We now explore possible ways to tackle some of the problems presented above.

Our first approach is to use a Database Layer. External data is stored into the database, in either fixed or temporary tables. The layer is responsible to understand the format which describes the data arising from external sources and add it to the database. Querying is then available to merge internal data with recently imported data. With this solution, developers could use Simple Queries and Advanced Queries (section 1.3.1) to achieve desired effect.

A Memory Layer is the second approach, where external data is held in main-memory in a general data exchange format like XML or an object-oriented representation. In order to merge internal and external data, a transformation needs to be performed between the format of internal data (relational data) and the now-memory-resident data format. The merge can afterwards be done with the help of an XML query language, such as XPath, or via a programming language like C#, through algorithms like nested loop join (section 3.5.1). All this transformation process must be invisible for a developer and it is performed during query processing.

The approach we effectively use takes the advantages of using both (and mix) a database and a memory layer. We build on the query language Linq (section 3.2.2) where we are able to specify queries between data from database tables and other data sources, such as web services. Merging is performed automatically in main memory and, by using optimizing algorithms and caching specific statistics, we achieve good efficiency results.

We consider the following criteria to evaluate and compare the advantages and drawbacks between the described approaches:

- Query execution time

- Space complexity
- Language's expressiveness improvement

Regarding to the database layer approach, query execution time depends on the time of inserting the data into the database and further merge query. Furthermore, a great amount of space is required to implement this solution. The database layer approach improves the expressiveness of the language because Simple and Advanced Queries could be used to access and transform such data. Finally, the next queries aiming the same data are retrieved much faster, since data is already stored in the database.

As for the memory layer approach, storing and merging the data in memory contributes with a speed-up in query execution time only if the map from relational data to the format of the data being held in memory is faster than uploading the external data into the database. This solution is more complex to implement and future queries have to deal with the same transformation since data is not cached locally.

Finally, our hybrid approach tries to join the advantages of both previous solutions. The process of fetching data from databases is done automatically by the underlying framework of Linq, which saves us time. Moreover, we do not need to store external results locally, in opposition to the first approach where all the information is saved. Hence, space complexity is not an issue. Linq has automatic caching mechanisms for queries and web services invocations and thereby future queries are performed faster.

The next table addresses the previous comparison, where "?" means unknown, "+" means good/better and "-" means bad/worse:

Table 1.1: Table addressing possible approaches

	1st execution	**Future executions**	**Space complexity**	**Expressiveness**
Database Layer	?	+	-	+
Memory Layer	?	-	+	-
Hybrid Layer	?	+	+	+

Given the presented criteria, it is understandable that the most suitable option for the considered criteria is an hybrid layer integration.

In order to specify the problem in detail and learn about concrete scenarios arising from daily routines in companies, we conducted a set of interviews to project managers who have contact with these integrations. From those, we extracted important criteria and understood which are the most discussed and problematic patterns when dealing with these integrations.

Our research focused not only on appropriated query languages for integrations with heterogeneous data sources, but also on query optimization techniques in relational databases and distributed systems.

Slow query executions tend to show up in systems dealing with large data sets and optimization techniques are therefore fundamental. Over the time, the dispersion and the size of data changes and the integration algorithms may become inefficient. Most of the times, developers lose time analysing the problem and manually changing the algorithms. Hence, an automatic process to implement these integrations is important.

1.5 RoadMap

We now describe the structure of the remaining of this document:

- Chapter 2 presents the different kind of data sources and, more specifically, the ones considered for this project. Further, we present well known web service providers typically used by every enterprise, like SAP and SalesForce. Finally, we end up with a scenario of a model containing some data sources that we will use constantly in this work, whether to present examples or to test and validate our solution.
- Chapter 3 presents a detailed study about the topics we researched for, in order to be able to design a solution for the problem presented. We write about query languages, multidatabases, query optimizations techniques over relational databases and distributed systems, and useful technologies that allowed us to develop a solution.
- Chapter 4 introduces the algorithm that allow us to execute the kind of queries we address. Still, we show which information needs to be gathered and a concrete example of the algorithm execution. Besides, we also present the model that supports the algorithm. We explain how we calculate the metrics maintained over the data sources and how they affect query execution.
- Chapter 5 describes the main implementation parts of our model and our algorithm.
- Chapter 6 presents the results achieved with our solution and the validation for the same. We tested our solution with developers of *OutSystems* in order to demonstrate that our solution is useful, automatic, and efficient.
- Chapter 7 ends this document with a conclusion over this work. Moreover, we formulate a proposal to the *Agile Platform*, where we explain what should be added to their model so our work can be integrated in the platform. Finally, we refer which topics will be address as part of future work.

2

Context analysis

In this chapter we make clear what specific problems we are researching and also describe which kind of data sources we are considering. Still, we present a study of web services examples provided by well-known providers (SAP and SalesForce).

2.1 Data sources

Different query languages have been developed over the time for the various types of data sources, for example, SQL for relational databases and XPath for XML. Data sources can range from a big variety of formats. There are structured sources, such as DBMS[1] (MySQL, PostgreSQL, Oracle, etc...), semi-structured sources (XML files, MS Excel files) and unstructured sources like flat files or html pages. We focus on the structured sources web services and databases.

Databases: In this context, a database consists in a repository containing relational data. They are the most used approach to store information and they can carry large terabytes of data. Databases can retrieve information when requested by a query. There are several database management systems, from which we focus on the relational ones, such as Oracle, MS SQL Server or MySQL. Inside a database, data is stored in structures called tables. A table can have many columns, each of them representing an attribute, while all the rows represent the saved data. We use the name row, record and tuple interchangeably in this document.

[1] Database Management Systems.

Web Services: In this context, a web service is a software system designed to support interoperable machine-to-machine interaction over a network [sitd]. A web service receives an input set I of data and retrieves an output set O. Input I can be composed by an attribute or a set of attributes, while output O is every time a set of records, which can be empty. Alternatively, a web service can be seen as a database with more restrict capabilities. Web services use the XML format to transfer data between its clients and themselves through a standard TCP/IP protocol, such as HTTP. Web services are structured data sources and they provide such information in a schema explaining how they work and which data they provide. Such schema also specifies how the service should be accessed, how many methods a web service provides, the parameters (cardinality and types) in each method call, among other details.

2.2 Web Services Examples

Web Services can provide a wide interaction infrastructure, depending on whether they offer many public methods or not. The collection of methods that can be invoked in a web service is called an API. Between the many services available for public usage, we show how the following ones work, as well as which API methods they provide. These are very well known external services and they are used by the majority of the companies, reason why they are relevant.

2.2.1 Salesforce

Salesforce[2] is a CRM[3] software enterprise integrated in the SaaS (Software as a Service) market, that grants efficiency and consistency to companies, by controlling over routine activities, eliminating redundant tasks, and automating business processes. Salesforce is a popular workflow automation engine for the full range of sales management needs. Companies can make smarter decisions about where to invest and show the impact of its marketing activities.

Salesforce offers access to the information of a company using easy, powerful, and safe application programming interfaces. Using SalesForce SOAP API[4] one can create, retrieve, update or delete records, such as accounts, opportunities, leads, or products. The API also allows the maintenance of passwords and perform searches.

Client applications can invoke these API calls to determine which objects have been updated or deleted during a given time period. These API calls return a set of IDs for objects that have been updated (added or changed) or deleted, as well as the timestamp indicating when they were last updated or deleted. It is the responsibility of the client application to process these results and incorporate the required changes into the local

[2]http://www.salesforce.com/
[3]Customer Relationship Management
[4]http://www.salesforce.com/us/developer/docs/api/

copy of the data. Therefore, from SalesForce API calls for data replication, we retain the following:

- query(*string queryString*) - *queryString* is a string specifying the object to query, the fields to return, and any conditions for including a specific object in the query. This API executes a query against the specified object and returns data that matches the specified criteria, caching the results of the query on the API. The query result object contains up to 500 rows of data by default. If the query results exceed 500 rows, then the client application should use the "queryMore()" call to retrieve additional rows in 500-row chunks. It is possible to increase the default size up to 2.000, as described in "Changing the Batch Size in Queries"[5]. Queries taking longer than two minutes to process will be timed out. For timed out queries, one must change the query to return or scan a smaller amount of data.

- queryMore(*QueryLocator queryLocator*) - *queryLocator* is an object containing a value that will be used in the subsequent calls for this API. This API retrieves the next batch of objects from a query() call. The "query()" call retrieves the first 500 records and creates a server-side cursor that is represented in the *queryLocator* object. This method processes subsequent records in up to 500-record chunks, resets the server-side cursor, and returns a newly generated QueryLocator.

- queryAll(*string queryString*) - Retrieves data from specified objects, whether they have been deleted or not. It is commonly used to identify the records that have been deleted because of a merge or a deletion. This API has read-only access to the field *isDeleted*, otherwise it is the same as *query()*.

- retrieve(*string fieldList*, *string sObjectType*, *ID ids[]*) - Retrieves one or more objects based on the list of fields to retrieve *fieldList*, the object type *sObjectType*, and an array of record IDs to retrieve (*ids*). This call does not return records that have been deleted.

- merge(*MergeRequest[] mergeRequests*) - *mergeRequests* is an array containing the objects to be merged. This API merges up to three records of the same object type into one of the records, deleting the others, and re-parenting any related records. Each merge operation is within one transaction and a batch merge has multiple transactions, one for each element in the batch. The only supported object types are Lead, Contact and Account. Any merge request has some limits, related to maximum merge requests in a single SOAP call and limit of records to be merged in a single request.

- update(*sObject[] sObjects*) - Updates one or more existing objects (*sObjects*). Client applications cannot update primary keys, but they can update foreign keys. For

[5]http://www.salesforce.com/us/developer/docs/api/Content/sforce_api_calls_soql_changing_batch_size.htm

example, a client application can update the "OwnerId" of an "Account", because "OwnerID" is a foreign key that refers to the user who owns the account record.

From SalesForce standard object's data model, which is considerable large, there are several entities such as accounts, contacts, opportunities and leads. Focusing on the object Opportunity, which represents a sale or pending deal, it can be used to manage information about a sale or a pending deal in a business context. A client can create, update, delete, and query records associated with an opportunity via the API, as well as update opportunities if it has an "Edit" permission for that. Opportunity object has information like:

- AccountId: a reference to an object Account.
- Amount: Estimated total sale amount.
- CampaignId: a reference to an object Campaign.
- CloseDate: required attribute representing the date when the opportunity is expected to close.
- Description: a text description of the opportunity.
- Name: required attribute representing the name for the opportunity.

2.2.2 SAP

SAP AG[6] is an ERP[7] that provides enterprise software to manage business operations and customer relations. They are the market and technology leaders in business management software, solutions and services for improving a business process. From their products we focus on SAP BusinessObjects software, which provides access to the information of a company. This software offers features to do reports and analysis, dashboards[8], data exploration, between others.

SAP offers service-oriented architecture (SOA) capabilities in the form of web services that are tied around its applications. It is organized into modules, each one representing a concept. Inside each module there several sub-models, each one representing a more specific sub-domain, until reaching individual objects. For instance, "Purchase Order" object type is under the model "Purchasing", which is under the sub-model "Materials Management", which is a sub-model of a global module "Logistics". As "Logistics", there are also the global modules "Financials" and "Human Resources", each of them containing a vast tree of sub-models inside. In the end, each object type will have a BAPI[9] with standard operations that we describe next.

[6] http://www.sap.com/index.epx

[7] Enterprise resource planning system that integrates internal and external management information over an entire organization, regarding finance, accounting, customer relationship management, between others.

[8] A graphical presentation of the current status of an application, for instance, regarding several criteria.

[9] Business Application Programming Interface

BAPIs of SAP Business Object Types: Remote function calls (RFCs) that represent an object-oriented view of business objects, enabling developers to perform integrations between the data from an enterprise and the data from SAP. The BAPI module accesses the corresponding method that applies to the object.

For example, the RFC module "BAPI_USER_GET_DETAIL" implements the "GetDetail()" method for the business object "User".

SAP BAPIs enable the integration of components and are therefore a part of developing integration scenarios where multiple components are connected to each other, either on a local network or on the internet. There are some standardized BAPIs that can be used for most SAP business object types. Standardized BAPIs are easier to use and prevent users having to deal with a large number of different BAPIs. Whenever possible, a standardized BAPI must be used in preference to an individual BAPI.

With object methods and especially with BAPIs, one can differentiate between instance methods and class methods. Instance methods refer to precisely one instance of an SAP Business Object type, for example, to a specific order, whereas class methods are instance-independent.

Standardized BAPIs to read data

- GetList() - selects a range of object key values, for example, company codes and material numbers. The key values returned by this BAPI can be passed on to another BAPI for further processing, for example, the BAPI "GetDetail()". Depending on each object, this method usually provides many input parameters (more than 10 in some cases), in order to allow an efficient information filtering process in SAP. A special input parameter which is not mandatory is "MaxRows", enabling one to limit the number of entries returned in a call.

- GetDetail() - retrieves the details of an instance of a business object type, identified via its key. Usually it has a few input parameters (3-5), enough to specify a unique object in SAP. Additionally, input parameters that determine the amount of detailed information displayed can also be specified. As an example of a call, for the business object "User" this BAPI call would retrieve logon data, default parameters, communication information, the user's company address and the user's assigned roles.

- GetStatus() - retrieves information about the status of a particular object instance. As for input parameters, they must contain the key fields of the corresponding business object type.

- ExistenceCheck() - checks whether an entry exists in the database for an SAP business object type, for example, whether the information exists within a particular company code for a customer. Moreover, this call can be used to check the existence of sub-objects at the same time, by providing optional parameters. For instance, the

input parameter "CompanyCode" in a BAPI call "Customer.ExistenceCheck()" will check the existence of a particular company code in the customer.

For further details about standard SAP BAPIs, consult this reference[10].

There are BAPIs where data transfer is huge. For some situations regarding purchase orders (PO), to call a BAPI "PO.GetDetail()" method, it is needed to supply around 20 input parameters and the results can have around the same number of output fields. From these input and output fields, many of them are complex structures composed by many attributes. In the end, the amount of data sent in an SAP call can achieve dozens or hundreds of records' attributes. In addition, an answer from a SAP call may encapsulate hundreds or thousands of records, each one made from dozens or even hundreds of attributes. SAP data model and its interaction patterns are in fact huge, not easy to deal with and not custom made. As examples of complex structures, consider an address containing information about cities, names, dates, postal code, streets and so on, or a purchase order containing information about dates, currencies, cashes, rates, agreements, and so on. These integration scenarios with SAP are real, ending up with large data transfer on issued calls.

Following, we present a concrete scenario with data from the data sources that were used to develop and test this project.

2.3 Scenario

We now present the data sources used during this project for testing our queries, validating and supporting our project. They consist of two database tables and three web services, addressing a scenario with samples of courts, judges working in courts, users and cities. The database contains courts and users, while the existing judges, cities and more detailed information about courts are available through web services. See Tables 2.1, 2.2, 2.3, 2.4 and 2.5 for a sample of these data sets, where *NIF* stands for the tax identity number. Consider, as well, the available web service APIs presented in Table 2.6. For each API, we specify its input arguments and its output cardinality, where *n* means that it may return several records and *1* means that it returns one record at most.

During the rest of the document we interchangeably refer these entities, provide many example regarding them and detail a little deeper its structure and other suitable information.

[10] http://help.sap.com/saphelp_46c/helpdata/en/7e/5e115e4a1611d1894c0000e829fbbd/frameset.htm

Table 2.1: Web service *WS_Courts*

Court_City	Court_Name	Court_NIF
Lisboa	LisboaComercio	234789511
Lisboa	LisboaJCriminais	734742111
Lisboa	LisboaPICivel	836564811
Aveiro	AveiroTT	134789511
Aveiro	Aveiro	353268447
Viseu	Viseu	600009270

Table 2.2: Web service *WS_Cities*

City
Almada
Setúbal
Faro
Guarda
Viseu
Castelo Branco
Leiria
Lisboa

Table 2.3: Web service *WS_Judges*

CourtName	JudgeName
LisboaComercio	Diogo Pires
LisboaComercio	Jorge Andrade
LisboaComercio	Andre Pereira
LisboaComercio	Miguel Nunes Silva
LisboaComercio	Joaquim Santos
AveiroTT	António Ismael
AveiroTT	João Almeida
AveiroTT	Paulo Guerreiro
AveiroTT	José Rui

Table 2.4: Database table *DB_Courts*

ID	CITY	NAME
1	Lisboa	LisboaComercio
2	Lisboa	LisboaFamilia
3	Lisboa	LisboaJCriminais
4	Lisboa	LisboaPICivel
5	Aveiro	AveiroTT
6	Porto	PortoJCiveis
7	Porto	PortoJCriminais

Table 2.5: Database table *DB_Users*

ID	NAME	EMAIL	ADDRESS	IDENTITYCARD	NIF	PHONE
1	Luis Antunes	la@gmail.com	Av. Fonseca de Matos	665744853	446587323	915888775
2	Antonio Serrate	as@gmail.com	null	1166653577	654778532	null
3	Garrick Luton	gl@gmail.com	null	21223441421	224784661	null

Table 2.6: Available API calls

WS_Courts	GetAll(): n GetByCourt_Name(*String name*): 1 GetByCourt_NIF(*int nif*): 1 GetByCourt_City(*String city*): n
WS_Cities	GetAll(): n
WS_Judges	GetAll(): n GetByCourtName(*String name*): n GetByJudge(*String name*): 1

3

Related Work

Services offered by companies typically depend upon other services and on the amount of data generally resident in external systems. Internet facilitates this cooperation and, as time flows, the time needed to send and receive desired data gets shorter. In spite of the technological improvements and the new ways of communication, integrating data from external services is still problematic.

The unlimited growth of data in databases generates efficiency problems for queries. Thus, efficient information storage is important and, as a consequence, finding desired data may be difficult and slow. Developers need to know which data they manage, their dimension and dispersion, where it is located and how to fetch it. Data can likewise arise different formats from several heterogeneous data sources and therefore data transformation processes may be needed. All these difficulties imply lots of concerns for developers that have to be extremely careful when developing algorithms to integrate data from several data sources, making more difficult its job. Moreover, the lack of mechanisms to automate these processes contribute as well, in a negative way, to the speed of projects development.

This section presents the research done on the topic of query optimizations, and it contains the base ideas for the development of our solution. Since query optimization is a well known topic for database systems, we decided to study how are query plans built and optimized, as well as the structures and statistics used to represent and optimize them, so then we can apply the same, or similar ideas to generate and optimize the execution plans for our queries.

3.1 Products

In order to turn integration scenarios automatic, tools have been developed for this special purpose. They have the acronym of ETL (Extract, Transform, and Load) tools and they extract data from internal and external sources by querying data relying on those systems. Data may then be transformed into a specific structure (required to proceed the operation), sorted, separated and so on. Finally, data is loaded into a data repository considering that it is desired to save it.

ETL tools: ETL tools aim to simplify data management by reducing the effort to produce data integrations. We tested some demos regarding many public tools to understand how and if they solve integration problems with data stored in internal and external sources. We found out that none of them allow to integrate databases and web services in a simple way, such as in a single query.

There is a private ETL vendor called Sesame Software[1] which owns a product that uses SalesForce Web Services API to process returned messages and it optimizes the interaction by making only the calls it needs. Further, where data is concerned, it uses an incremental replication method that only gets changed data from the last time the program has run against a particular object. It also provides options to configure the fields to be retrieved, so that only the needed is returned.

Their patent [BH12] consists in an incremental replication method which only gets data from a web service that has changed from the last time a query was made. This method results in a performance optimization because only fresh data is transferred from a web service, avoiding large data transfer and its related latency.

Many web service systems, such as SAP and SalesForce, already have configurable mechanisms to retrieve only fresh data from their systems and thereby we do not consider such synchronization features to be part of our solution.

3.2 Query languages

3.2.1 SQL

SQL [SKS10, sita] is a special-purpose programming language which allows developers to retrieve information from relational databases. Although, even referred as a query language, SQL can do much more than just query a database, since it allows not only to retrieve information but also to define the structure of the data, create and modify data in the database and manage data access control. Additionally, SQL also includes procedural elements.

SQL has reserved keywords that are divided in categories, from which we underline the following:

[1] http://www.sesamesoftware.com/

- DQL: Data Query Language - It is the most used part of SQL. Specifies a query with a description of the expected result. Command "SELECT".
- Clauses: modification conditions used to specify the desired data to select or to modify within a query. Commands: "FROM", "WHERE", "GROUP BY", "HAVING", "ORDER BY", "DISTINCT".
- Logical Operators: "AND", "OR, "NOT".
- Relational Operators: used to compare values in control structures. Some commands: "<", ">", "BETWEEN", "LIKE", "IN".
- Aggregation Functions: functions applied to a group of values that return a single value. They are used inside a "SELECT" clause. Functions: "AVG", "COUNT", "SUM", "MAX", "MIN".

For a complete list of SQL syntax categories, refer to [SKS10]. These are the operations presented in the queries we are considering. Hence, here follows an example of an SQL query, which retrieves all the courts of Lisbon from a table "COURT":

Listing 3.1: SQL query example

```
1  SELECT *
2  FROM COURT
3  WHERE COURT.City = "Lisbon";
```

3.2.2 Linq

Linq (**L**anguage **In**tegrated **Q**uery) is a Microsoft .NET extension that adds query features to some .NET programming languages. This language was incorporated into C# and Visual Basic with .NET Framework 3.5 and Visual Studio 2008, in order to provide queries over various data sources. Linq has a similar language syntax to SQL, allowing the construction of several instructions to extract information. It defines a set of functionalities based on query operands, lambda expressions and anonymous types.

Linq provides some different runtime infrastructures for managing different data sources:

- Linq to SQL[2] - for managing relational data as objects without losing the ability to query. It is designed especially to use in a data access layer, such as a database.
- Linq to XML[3] - provides an in-memory XML programming interface.
- Linq to Objects[4] - usage of Linq queries with any IEnumerable or IEnumerable$< T >$ collection directly. Instead of having to write complex "foreach" loops that

[2]http://msdn.microsoft.com/en-us/library/bb386976
[3]http://msdn.microsoft.com/en-us/library/bb387098.aspx
[4]http://msdn.microsoft.com/en-us/library/bb397919.aspx

```
var MERGE_QUERY = from dbCourt in dbCourts
                  join wsCourt in wsCourts on dbCourt.CITY equals wsCourt.City
                  select new
                  {
                      City = dbCourt.CITY,
                      NIF = wsCourt.NIF
                  };
```

Figure 3.1: Linq query example

```
//  DATABASE TABLE
CourtsDBDataContext dc = new CourtsDBDataContext();
var dbCourts = dc.OSUSR_D95_COURT5s;

// WEB SERVICE
QueryableCourtsServerData<WS_Court> wsCourts = new QueryableCourtsServerData<WS_Court>();
```

Figure 3.2: Data sources initialization

specified how to retrieve data from a collection, this infrastructure allows to write declarative code that describes what is desired to retrieve.

- Linq to DataSet[5] - makes it easier and faster to query over data cached in a DataSet object. It simplifies querying by enabling developers to write queries from the programming language itself, instead of by using a separate query language. It can also be used to query data that has been consolidated from one or more data sources.

When using Linq queries, one is always working with objects. The same coding patterns are used to query and transform data in SQL databases, XML documents, .NET collections, and any other format for which a Linq provider is available. According to [DC10], Linq queries over relational data sources are automatically converted to SQL by the underlying framework and sent to the database for the result.

For a wide set of Linq samples, check [Mic]. As a brief example, Figure 3.1 shows a Linq query merging data from a database table and a web service, where the data sources are initialized as shown in Figure 3.2.

The details regarding the initialization of the data sources, as well as other configurations that need to be done before one is able to write the previous kind of queries with Linq, are kept for section 3.6.

Query deferred execution: On scenarios like in the last example, Linq stores the view of the query in a data structure called expression tree, instead of immediately executing the query. This tree contains information about the table(s) the query is aiming to access, the question asked to the table, and the result that should be returned. Therefore, query

[5]http://msdn.microsoft.com/en-us/library/bb386977

```
foreach (var court in MERGE_QUERY)
       Console.WriteLine(court.Citv + " " + court.NIF);
```

Figure 3.3: Linq query deferred execution

```
var courtsQuery =   from c in dbCourts
                    select c;

var lisbonCourtsQuery = courtsQuery.Where(court => court.CITY == "Lisbon");
```

Figure 3.4: Extending a Linq query

execution is usually deferred until the moment when it is actually needed to request the data, for instance, when the output is iterated.

Deferred execution enables Linq to break queries into a relational algebra which makes composability possible, allowing developers to optimize their code. Furthermore, it makes possible to compose complex queries from several components without spending the time necessary to actually query such data.

Query execution is not always delayed. For cases using a call "Count()" or other operators that must iterate over the result of a query in order to return a specific value or structure, such as "ToList()", "ToArray()", or "ToDictionary()", query execution is immediate. In some situations, particularly regarding caching query results, it may be useful to force the execution immediately.

Linq deferred execution not only provides time saving by reusing queries whenever is necessary (instead of re-writing them), but also enables multiple queries to be combined or a query to be extended (composability). In the next example, the first query returns all the courts from the database and the second query extends the first by applying a "where" clause, aiming to return all the courts in Lisbon.

3.3 Federated SPARQL Queries

With the growing number of publicly available SPARQL endpoints [BQ08], federated queries become more and more attractive and feasible [RHpt]. Integrated access to multiple distributed and autonomous RDF data sources is a key challenge for many semantic web applications. As a reaction to this challenge, SPARQL, the W3C Recommendation [sitd] for an RDF query language, supports querying of multiple RDF graphs [BQ08].

However, the data sources we consider in our model are relational, therefore not following an RDF data format and thus federated SPARQL queries do not fit in our context.

3.4 Multidatabases

Multidatabases, also known as federated databases, are systems which transparently integrate several database systems into a single database. These databases are connected via a computer network and may be geographically spread. Multidatabases are an alternative to the laborious task of combining several disparate databases and enable the development of applications that need to access different kinds of databases [Kos00].

Such systems provide a uniform interface, allowing users to simultaneously access heterogeneous and autonomous databases using an integrated schema[6] and a single global query language [EDNO97]. To this end, a multidatabase system must not only be able to split a query into many subqueries, each one considering each database, but also compose the result sets of all the subqueries.

Typically, various database systems use different query languages, reason why multidatabases apply wrappers[7] to the subqueries, in order to transform them into the appropriate query languages.

3.4.1 Analysis

Considering the definition of multidatabases and an overview of the capabilities they offer, we now justify why they are not suitable to this project.

Even though multidatabases provide good capacities for data integration, they also offer difficulties which avoid us to adopt these techniques. Although multidatabases offer data integration techniques over several disperse databases, we are also interested in integrating information from external web services. To accomplish this feature, DBMS like MySQL, Oracle, MS SQL Server and DB2 have developed database-level components called engines to allow developers to write SQL queries over databases and web services. Such engines offer to developers a virtual view over the available data sources, which can be database tables and web services.

As an example, *Oracle JPublisher* [8] allows the extension of the storage, indexing, and searching capabilities of a relational database to include data sources like web services. The code is written in SQL, PL/SQL, or Java to run inside Oracle Database, which then calls the web services. To access data that is returned from invocations to web services, a developer must create a virtual table using a web service data source.

However, each DBMS has its own way to implement and use this feature. We aim to propose a less complex solution, adaptive over any data model, not focused in the data storage component.

[6] Single common data model

[7] Components that translate subqueries from internal formats into the query languages used by other data sources.

[8] http://docs.oracle.com/cd/B28359_01/java.111/b31225.pdf

3.5 Query optimization

Techniques to retrieve and load large amounts of data from the web into a relational database are usually fault-sensitive because of network related issues like data transfer speed and servers distance. Besides, web services have less performance and reliability than databases. Network delay and bandwidth capacity are the two most basic reasons for these constraints. Thus, it is important to optimize queries over distributed data.

We focus our study on query optimization techniques, more precisely on query plans optimization, statistics cache and cost estimation techniques. However, the optimization of these kind of queries is not a straight-forward process due to the fact that the system needs statistical information about the remote data, which typically is not available, and needs to optimize executions according to the web services APIs methods available, which may not be as rich as expected.

Before heading in the direction of non-centralized systems, we quickly explain how are the optimizations performed by relational database systems.

3.5.1 General concepts

Before facing the topic of query optimization in scenarios mixing remote and local sources, one should begin to understand how optimizations work in relational databases. To begin with, we summarize how database systems compute optimized plans, which algorithms they have available for usage, and which statistical information they rely on.

As presented in [Cha98], the two key components of a query evaluation component are the query optimizer and the query execution engine. A query optimizer receives a representation of a query and it is responsible for generating an efficient execution plan for the given query, passing it to the execution engine that executes the query plan.

Developers write queries in SQL and they are sent to the database system, which defines an execution plan and executes it. When a query is posed to a database, it may not be written in the most appropriate way, because developers may not have knowledge regarding how queries are executed. Hence, they can write queries that are slow to execute.

Database optimizers act before the execution of queries, in order to define a plan that is adapted to the underlying data. As referred in [SKS10], they rely on informations provided by its database catalog, which keeps statistical information about the database, such as the number of table records, the number of distinct records in columns, data histograms, and the number of disk blocks occupied by each table. A database catalog also keeps information about which indexes are available to be used. An index is a data structure storing data information (disk pointers, tuples), typically organized as a balanced tree, that aims at answering a query with high speed, avoiding disk accesses.

When data from a database table needs to be accessed, there are at least two ways of doing it. If no index is provided, the access is made through a sequential scan on the table

stored in disk. If there is an index on a relevant attribute, the optimizer may perform an index scan and only filtered tuples are accessed and retrieved.

The slower operations in databases are the join operations. Joins are typically slow because they may involve many entities and this operation implies scanning data from every entity and generating results by computing combinations of records. In those situations, for a good performance it is very important to have an efficient optimizer. There are several techniques available in this context, such as the hash join, merge join, nested-loop join, between some others and also variances between them. We now summarize the hash-join and the nested-loop join, presented in [SKS10].

- Hash join: it requires an equijoin ('=' operator on the join clause). The first phase (called build phase) prepares an hash table of the smaller relation. Entries of an hash table consist of the join attribute and its row. Once built the hash table, the second phase begins (called probe phase). Here, the larger table is scanned and the relevant rows are found from the smaller relation by looking in the hash table. This algorithm requires the smaller table to fit into memory, which sometimes does not happen. For those situations, there are modifications to the algorithm. The good performance of hash join resides in the fact that the hash table is accessed by applying an hash function to the join attribute. It is quicker to find the rows of an attribute by using an hash table, than by scanning the original relation.

- Nested-loop join: naive algorithm that joins two or more relations by making nested loops. The structure of the algorithm is as follows:

Listing 3.2: Nested-loop algorithm

```
1 For each tuple r in R do
2     For each tuple s in S do
3        If r and s satisfy the join condition
4           Then output the tuple <r,s>
```

Additionally, block nested loop join is a generalization of the simple nested loop algorithm that takes advantage of additional memory to reduce the number of times that the relation is scanned. It is thereby preferable to use it when it is available.

The performance of a query plan is determined mostly by the order in which tables are joined, as well as the appropriated join algorithms chosen. For example, when joining three relations "emp", "dept" and "job", with size of 20 rows, 200 rows and 2.000 rows, respectively, a query plan that performs "job join dept" at first place may be much slower than another joining "emp" and "dept", since it does not join the smallest collections at first place, thereby not reducing the output cardinality as it should. The algorithms performing all these kind of choices are more complex, but generally are summarized in two phases. First, all the ways to access each relation in the query are explored. Afterwards, the optimizer considers combining each pair of relations presented in a join condition. For each pair, the optimizer will consider the available join algorithms and preserve the

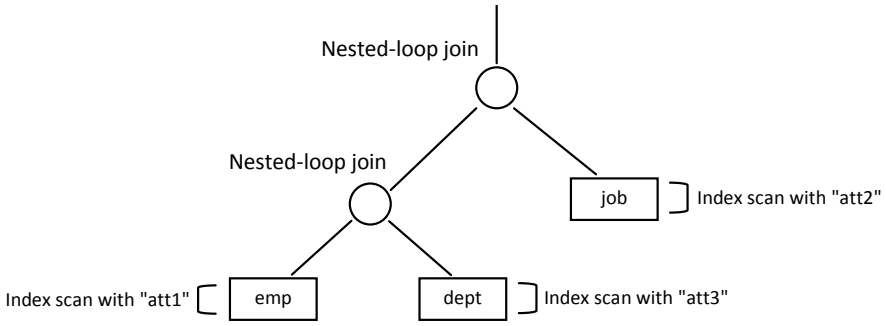


Figure 3.5: Query plan tree example

cheapest solution to join them. After, the query plans of all relations are computed by joining each two-relation plan produced by the previous phase with the remaining relations in the query.

These algorithms also pay attention to the sorting orders of the result sets produced by a query plan because particular sort orders may avoid redundant sort operations later on in processing a query. Therefore, a particular sort order may speed up a future join because it organizes the data in a particular way.

As an example of a query plan attempting to join data from "emp", "dept", and "job", with available indexes on relevant join attributes, see Figure 3.5.

For more optimization techniques regarding indexing, queries, and concurrency control, refer to [SBB03].

For scenarios where queries deal only with internal database data, the database query optimizer and further engine take care of the process. However, how can it be done if a query selects data from both internal and external sources, or only from external sources? The process becomes more complex since now the work cannot be done by a simple database query optimizer.

While in a relational database system there are available indexes and statistics, when considering external systems like web services, such concepts may totally disappear, issuing a great need of performance optimization. During query processing, it should be detected whether the query involves external sources or not. If the query fits in this case, splitting the query into several parts is a solution. The existence of an intelligent component capable of splitting this kind of queries in several parts, regarding several data sources, is useful. Queries in touch with database data are sent to the database and we do not need to worry about their execution, since the database management system already takes care of this issue. On the other hand, queries regarding web services have to transformed into API calls, which can be invoked and data is fetched. Moreover, result sets coming from both parts may have to be merged to compute joins. Nevertheless, how can a query plan over these heterogeneous sources be built, and how to optimize it?

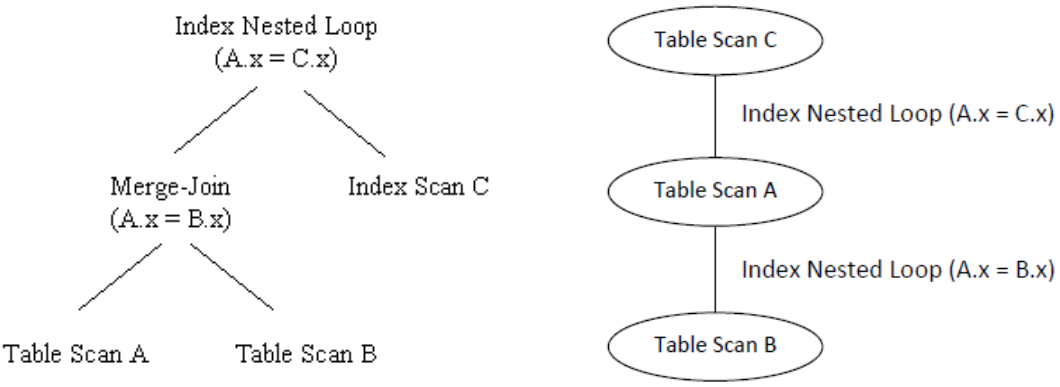


Figure 3.6: Query plans representation

Query plan structure: A query plan specifies precisely how the query is to be executed. Typically, every database management system represents a query plan as a tree [Kos00], and many studies in query optimizations use trees to represent query plans [SKS10, SO95, DSD95]. These trees can also be referred as physical operator trees [Cha98]. Each node of the tree represents an operator carrying out a specific operation through one or more sets of data, such as table scans, joins, group-bys, sorts, among others. These nodes are annotated with information representing, for instance, the expected size of the data resulting from that operation, where the operator is to be carried out, the cost of that operation, or other statistical information available [SKS10]. The edges/connections represent either the data flow of the execution or the dependencies between nodes.

Another representation of queries are graphs. The authors of [RGL90] address this scenario and they refer that every relation in the query is represented as a node, and that every join operator and predicate must be represented as well. Additionally, edges connect nodes. In [GSS07], the authors present an algorithm regarding the efficient execution of query plans, where the query is represented as a graph. [SMWM06] broadens the domain of query optimization by considering query plans over databases and web services, represented also as graphs. Figure 3.6 shows an example of two plans following the described structures. The left image represents a tree representation and the right image a graph representation.

Query plan transformations: In this project we only consider inner joins. Due to its commutativity [RGL90, Cha98], they can be freely reordered, raising the different execution plan possibilities. Therefore, several transformations may be applied to execution plans, in order to generate different representations that may result in different execution costs as well. [RGL90] presents some basic transformations that can be applied to query plan trees. The aim of these transformations is to find a better order for executing a sequence of joins.

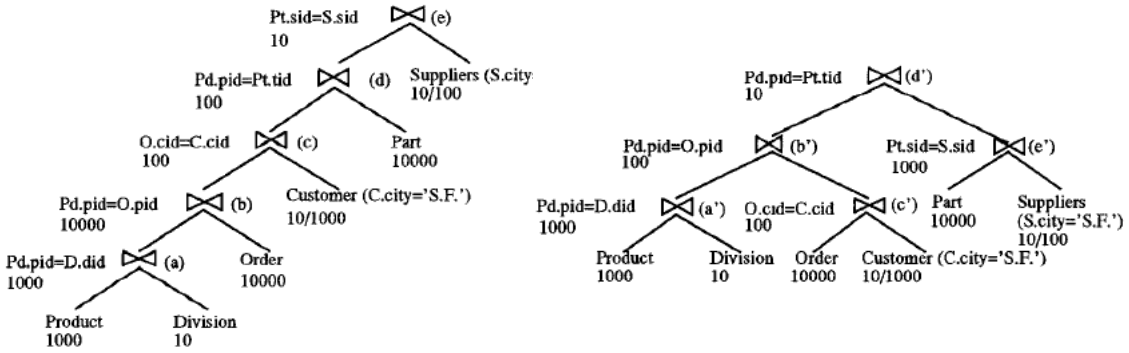


Figure 3.7: Tree balancing

In [DSD95] the authors show an example of a complex join query that can be represented by a left deep join tree (Figure 3.7, left part), and by balancing such tree (Figure 3.7, right part), a better overall cost is achieved. That example focuses on the idea that joins between entities that can be filtered by available predicates should be executed first, thereby reducing the soonest possible the result cardinality. In the end, when joins that cannot have its cardinality reduced are to be executed, a smaller amount of data is given as input and the overall join cost is smaller. Algorithms for tree balancing are also presented in [DSD95].

Grouping and sorting: Optimizations regarding these operators are presented in some of the referred articles, such as [DSD95], but since we are not focusing on grouping and sorting features, we place them as part of future work.

Join techniques: In join scenarios, when data is retrieved from entities, a memory join may have to be computed. To this end, there are several techniques that can be applied. To our case, a simple scenario is a join between a database table and data arising from a web service. Some of the existing algorithms to implement these joins are the naive nested loop join, the indexed nested loop join, the hash join and the merge join. All these algorithms are described in [SKS10]. For our case, since we only consider equi-joins on equalities, we choose to implement one of the efficient algorithms, the indexed nested loop join, applied in our solution when a merge in memory between two collections is necessary. Thus, we place the hash join and the merge join as joining techniques to be included in our solution in the future, better suitable for joins with complex conditions.

Statistics: Different execution orders typically have different execution times. It is therefore important to have cost estimate metrics to be applied to parts of the plan, for example to a join operation, to guarantee that the considered operation cost is computed and therefore reducing combinations, by not considering different ways of executing those operations. A successful optimizer must have reliable statistics to apply cost estimate

heuristics that allow to understand which plans are more efficient than others [SKS10]. There are three main fundamental statistics that must be maintained:

- total number of rows of an entity
- expected number of rows produced by an operation, also known as output cardinality
- expected time cost of an operation execution

The concept of selectivity appears in this topic and it is usually used as a measure to estimate the cost of an operation. The more selective an operation is, the less records it produces and therefore the better for future operations, since they will have to deal with less records. This term can be applied to queries and APIs, and to the columns of entities, as follows:

- Call selectivity [SMWM06]: number of retrieved rows, per given input. For instance, for an API *GetByCity(string city)*, if we invoke it and we obtain 2 rows, its selectivity is $\frac{2}{1} = 2$.
- Column selectivity [Ell02]: number of distinct values in a column divided by the total number of rows in the entity. For example, if we want to know the selectivity of a column *City* of an entity *Courts*, considering that we have 100 courts and there are 50 different cities, the selectivity of the column is $\frac{50}{100} = 0.5$.

Moreover, there are statistical measures over the columns of entities that should also be maintained, as covered in [GS07, Cha98, Ell02, SKS10]. These are some of very important summaries that help understanding the selectivity of certain predicates/filters applied on columns, and of join operations:

- number of distinct values in the column
- number of null values in the column
- column uniqueness
- column is a foreign key to another entity's column

We consider these to be the minimum required statistics to be maintained that allow us to optimize the join queries we tackle, between database entities and web services.

Cost estimation of filter predicates: [SKS10, Alf] describe how to compute the expected number of rows resulting from the application of filter predicates. In this topic, we present the related material and we already reveal how we estimate the cost of filters in our solution. For this exercise, consider the entities and the notations presented in Figure 3.5.1. Consider as well the available statistics over the entities:

A	
id	X
1	a
2	c
3	d
4	d
5	a
6	null
7	b

B	
id	Y
1	a
2	c
3	d
4	d

- n_A = total number of rows in A
- $nulls_{A.X}$ = number of nulls in $A.X$
- $dist_{A.X}$ = number of distincts in $A.X$
- $n_{A.X^*} = n_A - nulls_{A.X}$

Figure 3.8: Filter estimation example

- $n_A = 7$, $n_B = 4$
- $nulls_{A.X} = 1$, $nulls_{B.Y} = 0$
- $dist_{A.X} = 5$, $dist_{B.Y} = 3$
- $n_{A.X^*} = 6$, $n_{B.Y^*} = 4$

To estimate the expected number of rows of a filter predicate, we follow the next hierarchy of estimations. Every time an estimation is false or unknown, we continue to the next step, otherwise we compute the estimation and the sequence stops. Consider a filter $A.X =$ *"b"*.

1. If the column is unique, the filter estimation is 1 row.
2. If we have knowledge about the average number of rows retrieved by a query/call related with such filter, the filter estimation is that average.
3. If we know the number of distinct values in the column, the formula presented in [SKS10] can be applied to compute the size estimation for the filter:

$$\frac{n_A}{dist_{A.X}}$$

4. If we know the number of null values in the column, the estimation is the total number of rows of the entity minus the number of null values ($n_{A.X^*}$, for instance). This formula is not presented by the references and thereby is introduced by us.
5. Worst case: none of the previous steps succeed and the filter estimation is the total number of rows in the entity.

Following the sequence, X is not unique, we do not maintain an average of returned rows of a filter of this kind, but X has 5 distincts. Therefore:

$$\frac{n_A}{dist_{A.X}} = \frac{7}{5} = 1.4$$

When presenting the final result we always round the estimation to the nearest integer value, in this case 1.

As we can see, for this example, the estimation is precise since the column has 1 row with the value "b" and we estimate also 1. If the number of distincts for the column was not available, the best estimation we could do was to take out the number of null values in the column (if this measure was available) from the total number of rows in the entity. In such case, the estimation would be $n_{A.X^*} = 6$.

For a set of filters applied to an entity, [SKS10, Alf] also present an estimation technique for the expected number of rows retrieved. Assume this set of filters applied to entity A, as well as its size estimation:

- X = *"b"*, estimation = 1.4
- *id* = *"2"*, estimation = $\frac{7}{7} = 1$

The following formula can now be applied to compute the estimation, where $s_1, s_2,$ and so on represent the size estimation for the filters:

$$n_r \times \frac{s_1 \times s_2 \times \cdots \times s_n}{n_r^n}$$

Thereby, for this example the estimation obtained is:

$$7 \times \frac{1.4 \times 1}{7^2} = 0.2$$

Once again, for the example considered the estimation is precise, since the result produces 0 records and we estimate 0 as well.

Cost estimation of joins: [SKS10, Alf] also describe how to compute the expected number of rows resulting joining two entities, regarding some available statistics. Consider again the entities, notation and statistics presented in the previous exercise, plus the join result between them, in Figure 3.9.

The worst estimation possible, applied when no statistics are available, is the cartesian product of the size of both entities. For this example, that would be $n_{A.X} \times n_{B.Y} = 28$ rows. Nonetheless, we may estimate much better. This is the hierarchy of verifications we follow when estimating the output cardinality of a join:

1. If both columns presented in the join condition are unique, the estimation is the minimum of the total number of rows of the entities [SKS10, Alf].

A	
id	X
1	a
2	c
3	d
4	d
5	a
6	null
7	b

B	
id	Y
1	a
2	c
3	d
4	d

A inner join B on A.X = B.Y	
A.X	B.Y
a	a
c	c
d	d
d	d
d	d
d	d
a	a

Figure 3.9: Join estimation example

2. If only one column is unique, the estimation is the total number of rows of the other entity [SKS10, Alf].

3. If one column is a foreign key to the other column, the number of rows produced is exactly the total number of rows on the entity having the attribute that is a foreign key [SKS10, Alf].

4. If the number of distinct values on both attributes are available, we can apply the formula presented in [SKS10, Alf] to compute the estimation:

$$\min\left(\frac{n_{A.X} \times n_{B.Y}}{dist_{A.X}}, \frac{n_{A.X} \times n_{B.Y}}{dist_{B.Y}}\right)$$

5. If the number of null values on both, or in just one column is available, we can compute the product of the sizes of the entities, taking out these null values, since they do not appear in the result set. Therefore, we subtract them from the total number of rows in the entity and afterwards compute the product of these values. This is the same as $n_{A.X^*} \times n_{B.Y^*}$, for this example.

6. Worst case, the estimation is the product of the sizes of the entities. For this example, that would be $n_{A.X} \times n_{B.Y}$.

Following the sequence:

1. We do not have uniqueness measures regarding the columns involved in the join

2. We do not know if any column is a foreign key

3. We know that $dist_{A.X} = 5$ and $dist_{B.Y} = 3$. By applying the formula presented in [SKS10, Alf], we obtain:

$$\min\left(\frac{n_{A.X} \times n_{B.Y}}{dist_{A.X}}, \frac{n_{A.X} \times n_{B.Y}}{dist_{B.Y}}\right) = \min\left(\frac{28}{5}, \frac{28}{3}\right) = \frac{28}{5} = 5.6$$

In this case, the estimation is not perfect but it is still precise, since the join produces exactly 7 rows and the estimation is 6.

These kind of joins have a single condition. However, how can we estimate the output cardinality of joins with several join conditions? The references presented do not consider these scenarios and therefore we implemented a simple metric to estimate these costs, since our solution computes joins with several predicates on a condition. The approach we use to compute an estimation of the number of rows produced by a join like

$$A = B \ \& \ B = C \ \& \ ...$$

is to compute, for each expression, the estimation we described previously and, in the end, take the minimum cost as the result for the join estimation.

Join reordering: [SMWM06] presents an algorithm to compute the optimal order of web service invocations in a plan, for queries concerning database and web service entities. The authors rely on two statistics: the selectivity and the time cost of web service calls. Nonetheless, their approach has a limitation because it considers that data from databases is fetched before invoking a sequence of web services. This problem is addressed in section 3.6.1.1, where we show that for certain scenarios this is not an efficient approach and therefore we do not follow the authors work.

On the other hand, [GSS07] presents an application of the Kruskal's algorithm to query optimization. Refer to [Mamb] for the explanation of the algorithm. Basically, the algorithm needs the queries to be represented as graphs. Arcs connecting nodes are joins between entities, while nodes are entities. Furthermore, arcs have costs, like its expected time cost, its expected number of rows, or another metric considered. Having such a graph built, Kruskal's algorithm computes the minimum spanning tree of the graph, which in this case represents the best join order, with minimum cost produced at each step. [Mamb, Mama] support the algorithm by presenting several data structures and complexity studies regarding the implementation of graphs data structures and Kruskal's algorithm.

After the research done on these articles we learnt and gathered the important concepts behind query optimization, as well as several techniques that should be applied to achieve efficient execution times. Thus, we propose to build query plans over databases and web services, with relevant information annotated (conditions, filters, cardinalities, costs, and so on), and maintain statistical information over entities and columns to enrich the model and rise the probability of building efficient query execution plans.

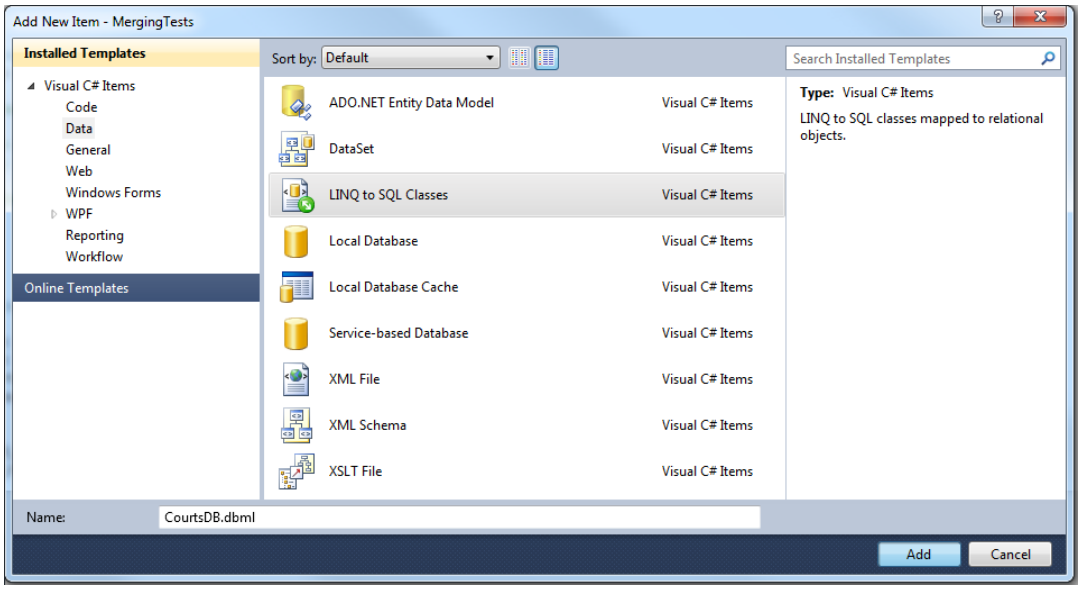


Figure 3.10: Adding a database data source

3.6 Technologies

In this section we describe the technologies we studied that have purpose for this project, therefore allowing developers to integrate information from disparate data sources.

3.6.1 Linq

As presented in section 3.2, Linq is a programming language the allows developers to write queries that are able to merge data from several data sources. However, due to the more complex scenarios developers can develop with it, Linq is also considered a technology. While before we described the features of Linq as a programming language, now we describe Linq as a technology where it is possible to build complex projects containing connections to data sources, and also configure them so a developer is able to write queries that automatically merge data from them.

We used Visual Studio 2010 to build these projects using Linq and thereby all the presented screen shots come from its environment.

Creating a database connection with LINQ: To begin with, we describe how a developer can add database entities to a Linq project. A database entity is ready to be instantiated when a developer adds a Linq-to-SQL data source to the project. The database data source creation step is represented in the Figures 3.10 and 3.11.

In these last figures, the entities are added to the database context data source simply by dragging available tables, which are inside the server connection "Tables" selector. Only the entities dropped in the context graphic area will be available to be queried with

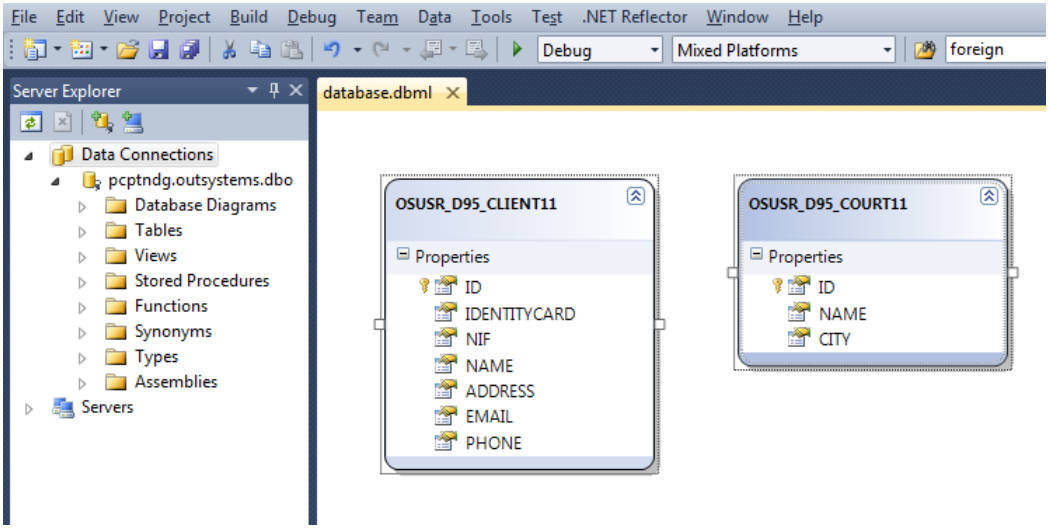


Figure 3.11: Adding a queryable entity to the database data source context

```
databaseDataContext dc = new databaseDataContext();
var dbCourts = dc.OSUSR_D95_COURT11s;

var db_query = from dbCourt in dbCourts
               select dbCourt.CITY;
```

Figure 3.12: Querying a database with Linq

Linq. After these steps are done, we can initialize the data source and write queries like the one presented in Figure 3.12.

Creating a Web Service connection with Linq: In order to allow querying web services with Linq, it is required to go through a set of steps that we summarize now. For a detailed description, check the very well documented entry in microsoft msdn library[9].

In a simple way, one has to develop a component that parses a Linq query and interacts with the web service. This component is called provider. Basically, a provider handles a kind of query and expects certain information to be specified. It has a closed type system, exposing a single type to represent the result data. The steps needed to create the web service provider are described in the appendix chapter A.1.

The component needs to be included in an application and, by the end of this task, queries against web services can be written. It is also needed to initialize the component before querying the web service, as shown in Figure 3.13. Afterwards, queries like the one presented in Figure 3.14 are ready to be written.

This provider examines only one type of expression in the expression tree that represents the query: the innermost call to "where". It extracts the data that it must have from

[9]http://msdn.microsoft.com/en-us/library/bb546158.aspx - last checked on 18-03-2013

```
QueryableCourtsServerData<Court> wsCourts = new QueryableCourtsServerData<Court>();

var WS_QUERY = from court in wsCourts
               select new
               {
                    City = court.City,
                    NIF = court.NIF
               };
```

Figure 3.13: Querying a web service

```
var MERGE_QUERY = from db_court in dbCourts
                  join ws_court in wsCourts on db_court.CITY equals ws_court.City
                  where ws_court.City == "Lobnya" || db_court.CITY == "Zverevo"
                  select new
                    {
                       City = db_court.CITY,
                       NIF = ws_court.NIF
                    };
```

Figure 3.14: Merging query

the expression, in order to query the web service, and then calls the web service APIs, inserting the returned data into the expression tree in the place of the initial *IQueryable* data source. The rest of the query execution is handled by the implementation of the standard query operators, such as the join operator.

All these queries are automatically executed by Linq. In the previous case, the execution is as follows: first, Linq fetches data from the database, taking into account the information supplied in the where clause regarding to the database entities. Afterwards, it queries the web service, again taking into account the information supplied in the where clause related to that web service.

Nevertheless, what if this execution flow is not the more efficient? Actually, there are cases where it is faster to query the web service before querying the database, in join queries similar to the previous one, as we show next.

3.6.1.1 Limitations

The query execution algorithm of Linq is hard-coded and cannot be changed. It is possible to implement some extensions, such as creating new functions that receive data from the query and apply some transformations, but it is not possible to extend nor change the query execution order, neither standard operators like the join.

In order to explain the limitations of these Linq providers to our project, we now present a concrete scenario, showing some queries and comparing their different execution order efficiency. Recall the scenario presented in section 2.3. To exemplify the efficiency contrast with different execution orders of a query plan, consider the query

```
var query = from ws_court in wsCourts
            join db_court in dbCourts on ws_court.Name equals db_court.Name
            where db_court.Name == "SantaremTT"
            select new {
                ws_court,
                db_court
            };
```

Figure 3.15: Test query

presented in Figure 3.15.

Linq executes the following query by calling the web service first, since it respects the order of the join operator. Thus, because we do not have any filter regarding the web service entity in the where clause, no filter information is available at this step and the API call *GetAll()* is the one invoked, retrieving all the available web service data. Afterwards, Linq queries automatically the database, applying the available filter, which retrieves one record and, in the end, the join is computed.

Now, consider the opposite pipeline execution. If we query the database first, we get a single record which can be used to invoke a *GetByName(string name)* API call. This call returns a single record and the final result can be retrieved. Such distinct execution order results in an efficiency improvement because data transfer is extremely reduced. Instead of invoking an external *GetAll()* API call that retrieves all the records, we could just invoke a fast indexed call as *GetByName(string name)*, therefore minimizing the data transferred in the network. With a small data set like the one we present, considering the fast technology we face nowadays, both query executions would be fast. However, in real companies' applications and information systems, the amount of data they deal with is never small, reason why it is fulcral to tune query execution.

In addition, there may occur other situations where query execution could be optimized, but with Linq providers like the one presented, that is impossible. In certain merge scenarios, considering that firstly the data is fetched from a database and after a web service is invoked, in order to understand which API call is the best to invoke, we need to analyse the data returned from the database, so we can apply some optimization metrics. Thereby, consider the next query:

Consider that it is more efficient to query the database first. Then, before querying the web service, there are two possible solutions:

- Invoke *GetAll()* and join the data afterwards
- Invoke *n* * *GetByName(string name)* and incrementally build the joining result, where *n* is the number of records retrieved from the database

```
var query = from db_court in dbCourts
            join ws_court in wsCourts on db_court.Name equals ws_court.Name
            select new {
                 ws_court,
                 db_court
            };
```

Figure 3.16: Test query

Consider as well that it takes 200ms to invoke the *GetAll* API, while to invoke a *GetByName* takes 20ms. Thus, if the number of records retrieved from the database is, for instance, 5, it is more efficient to invoke 5 times a *GetByName*, which incrementally builds the join result and no further memory merge is necessary, instead of invoking the *GetAll* API which takes longer and it implies a memory merge of collections.

As we can see, the impossibility to analyse the set of available data before invoking a web service, as well as not being able to define an algorithm that decides the query execution pipeline, are limitations that prevent us to use this technology to develop our project, since we cannot implement the optimizations we aim to.

3.6.2 Re-Linq

Although it is possible to create a Linq provider for a specific data source, as we presented before, creating these providers for web services is not a simple task, due to the amount of classes, parsing and code needed, plus the difficulty from a provider to analyse and understand the structure of the AST generated by the compiler, for a given Linq query.

Re-Linq foundation [Sch] was born to simplify these problems. Re-Linq is a framework that implements the difficult parts of parsing and understanding the ASTs generated by Linq query expressions once and to be reused for any purpose. Thus, it is not a provider nor an O/R mapper. Its goal is to provide a semantically rich and organized model of a Linq query, in a way that other Linq providers can take that model to build and execute their queries. Figure 3.17 illustrates the architecture.

To better understand what Re-Linq does and how it represents a Linq query, we introduce its transformation process. First, it analyzes a generated Linq AST and builds a *QueryModel* which holds instances of *SelectClause*, *MainFromClause*, *AdditionalFromClause*, *LetClause*, *OrderByClause*, *GroupByClause*, and other clause objects. Second, it analyses the expressions used by the different clauses and builds a data model for each of these expressions linking the values being used with their originating clauses, resolving property paths and partially evaluating those expressions that do not involve external data. Finally, the *QueryModel* is sent to a query executor, which must be supplied by the specific Linq provider. This query executor holds the execution algorithm for the supplied

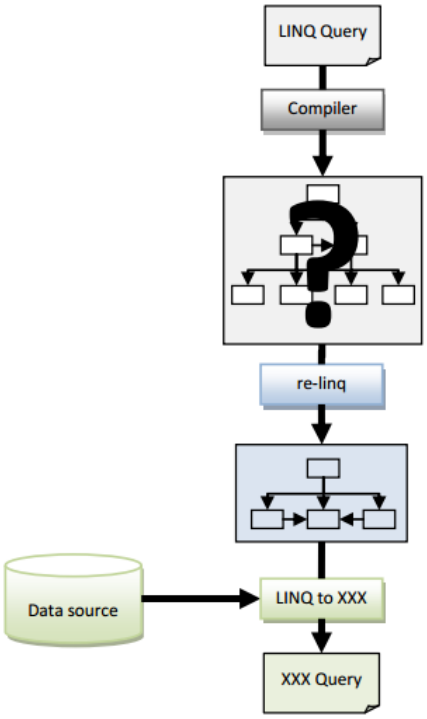


Figure 3.17: Re-Linq model, [Sch]

model. For our case, it generates queries to be executed in both databases and web services, execute them, and perform the necessary in-memory operations. Figure 3.18 shows how the next query is represented in a *QueryModel*:

Listing 3.3: Original Linq query

```
1 from c in QueryFactory.CreateLinqQuery<Customer> ()
2 from o in c.Orders
3 where o.OrderNumber == 1
4 select new { c, o }
```

3.6.2.1 Creating a Linq provider with Re-Linq

Since we want to write a Linq provider, we need to implement the necessary interfaces. An implementation of *IQueryable<T>* is needed because it contains the main query interface of Linq, and all of its query methods are written against it. Re-Linq provides a base class, *QueryableBase<T>*, from which one can derive to implement that interface. To do that, it requires adding two constructors: one used by the users of the provider and another used by the Linq infrastructure in the .NET framework.

Then, we need to implement the interface *IQueryProvider*. Linq query methods use this interface to create new queries around an existing *IQueryable<T>* and to actually execute queries. However, *QueryProviderBase* abstract class already has a default implementation, *DefaultQueryProvider*, which implements the *IQueryProvider* interface and thereby

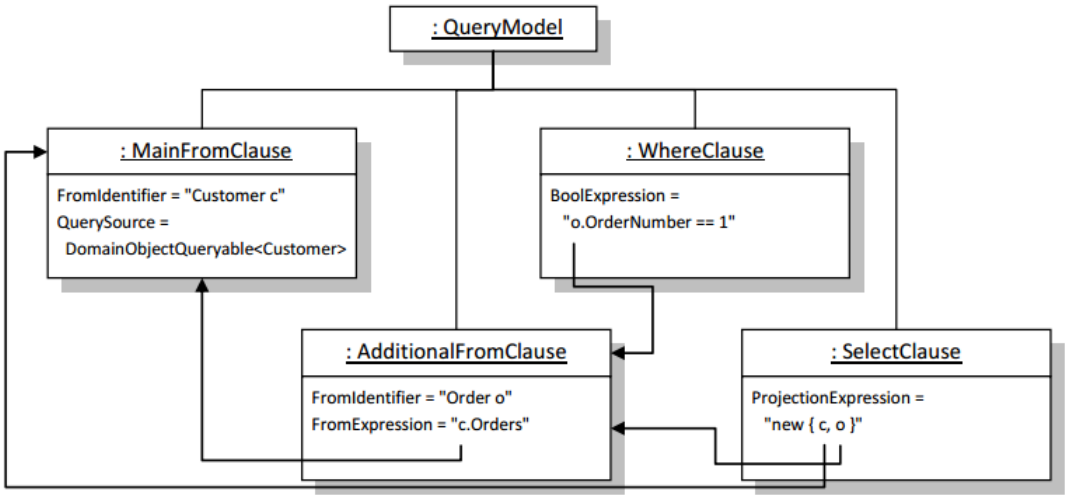


Figure 3.18: Re-Linq Query Model, [Sch]

we do not need to add any code, since *QueryableBase<T>* uses that implementation by default. While *DefaultQueryProvider* implements the query creation part of *IQueryProvider*, it cannot implement the execution of a query. Instead, it does the following:

- It parses the query which is to be executed, into a *QueryModel*
- It passes the *QueryModel* to an implementation of *IQueryExecutor*

These previous steps are illustrated in Figure A.4, located in the appendix, where we start the creation of our provider called *"ProviderQueryable"*.

IQueryExecutor is an interface representing the details of executing a query against a target queryable system and thus containing the query execution algorithm. It needs to be implemented by us, since we are the ones knowing how to build queries against our target data sources. We implement the execution for *"ExecuteCollection<T>"* method, since this is the one invoked for the class of queries we deal with. Our queries may return 0 or more records, but they will always invoke *"ExecuteCollection<T>"*. This last step is shown in Figure A.5, located in the appendix.

After these steps are performed, one can start the development of the query provider.

3.6.2.2 Limitations

In spite of offering to developers a higher starting level for the creation of Linq providers, this solution still demands a great amount work, since we aim to create a new query optimizer and thereby we have to implement all the code, ranging from the parse of queries to their execution using different data sources.

Re-Linq is referred by its authors to be focused for developing a Linq provider for any (but a single) data source, and not for many. However, with this project we experimented the opposite because we could easily separate the execution of queries against databases

and web services, once the appropriate query parsing was done and stored in a consistent and organized model.

Moreover, due to the novelty of this work, its documentation is still basic and our work plan started by getting used with the framework and start to implement some basic features, such as:

1. Check whether a query involves more than a single data source
2. Find the type of a data source (database table or web service)
3. Execute a database related query directly, through *Linq-To-SQL*
4. Execute a web service call, through an available API

In the next chapter we introduce the solution we developed for this dissertation.

4

Query execution

We present an algorithm that allows to efficiently execute queries ranging simultaneously over databases and web services by choosing the best execution flow, choosing the path with less estimated cost, and computing the minimum number of records at each iteration. Besides, by being supported by a model that maintains performance metrics over the data sources, its attributes, and queries executed, the algorithm executes the most efficient queries/APIs at every execution step.

This algorithm automates a process that is manually performed by developers when integrating data from database tables and web services: the manual implementation of a data retrieval algorithm. In addition, it follows an optimized and adaptative execution plan.

As we present next, we do not compute a best execution plan and execute it. Instead, the execution plan starts by choosing a starting point and then execute it, adapting the remaining plan based on previous results. Therefore, we mix both an optimizer and an executor as the component responsible for executing queries.

All the graphs presented were built with *GraphViz* tool [1]. In order to easily produce all the graphs in this dissertation, we implemented a tool that dumps a query graph directly to *GraphViz* syntax. We follow the legend in Figure 4.1 to represent our graphs.

4.1 Execution algorithm

In our algorithm we represent query plans as graphs. This algorithm is inspired in [GSS07], although it has some improvements and modifications.

[1] http://www.graphviz.org/

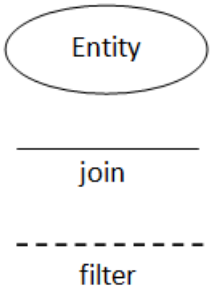


Figure 4.1: Graph legend

```
from wsCourt in courtsWS
join dbCourt in courtsTable on wsCourt.Court_Name equals dbCourt.NAME
join wsJudge in judgesWS on dbCourt.NAME equals wsJudge.CourtName
where wsCourt.Court_Name == "LisboaTT"
select dbCourt;
```

Figure 4.2: Merging courts and judges

We now provide an example of a query and its representation as a graph to explain the optimizing query engine. Thus, consider the query represented in Figure 4.2 that merges data from the entities *DB_Court*, *WS_Court*, and *WS_Judge*, presented in section 2.3. Moreover, consider the graph representation for the query in Figure 4.3.

There are two basic structures in the query graph: nodes and arcs. A node represents an entity, which can be either a database table or a web service. An arc either connects two nodes (representing a join), or connects a node to an expression being applied to that node (representing a filter). Nodes have several annotated information about the entity that is used during the execution algorithm:

1. the name of the entity
2. dimension: total number of rows in the entity
3. expected rows: expected number of rows resultant from the application of all filters to the entity
4. output: the minimum set of attributes needed to be maintained in the data set

There are two kinds of arcs in the graph: joins and filters. A join connects two nodes and it contains a join condition. On the other hand, a filter connects a node to an expression. Both kind of arcs have annotated the expected number of rows resulting from that operation.

The execution follows the arcs with minimum cost on each step of the query plan. The execution is adaptive because after each iteration, the optimizer updates de remaining

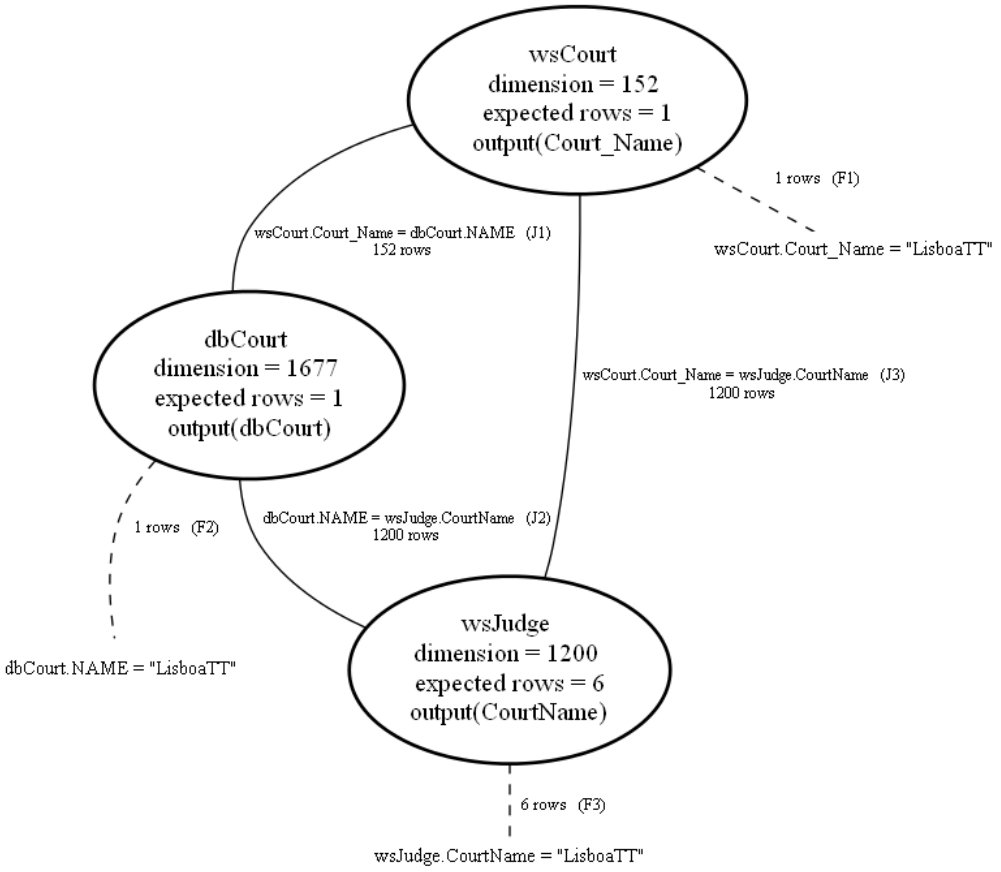


Figure 4.3: Graph representation

operations in the queue with new and more precise costs that may change the order of the queue, and therefore the remaining execution order. Hence, the algorithm is supported by a priority queue of arcs, ordered by minimum costs. We represent each arc in this queue by its label presented in the graph. For this graph, such queue would contain the order *F1*, *F2*, *F3*, *J1*, *J2*, *J3*.

The optimizer performs a loop over the elements available in the queue, until the queue is empty, which means that all the parts of the query were executed. On each iteration, an arc is removed from the queue and executed. If a removed arc is a filter, the optimizer removes all the remaining filters being applied to that entity from the queue because when the optimizer fetches data from an entity, all the available filters can be applied on the new data set, reducing it at the most. Therefore, when the optimizer fetches data from an entity, by executing a query or invoking an API, all the filters available over that entity are applied. On the other hand, when a join arc is removed from the queue, the optimizer executes that join by fetching the data that is not yet available and computing the join of both collections. Once again, the optimizer removes from the queue all the filters that are being applied over entities that are involved in the join arc removed. Details concerning how the optimizer executes filter and join arcs, as well as how the memory joins are computed are described on chapter 5.

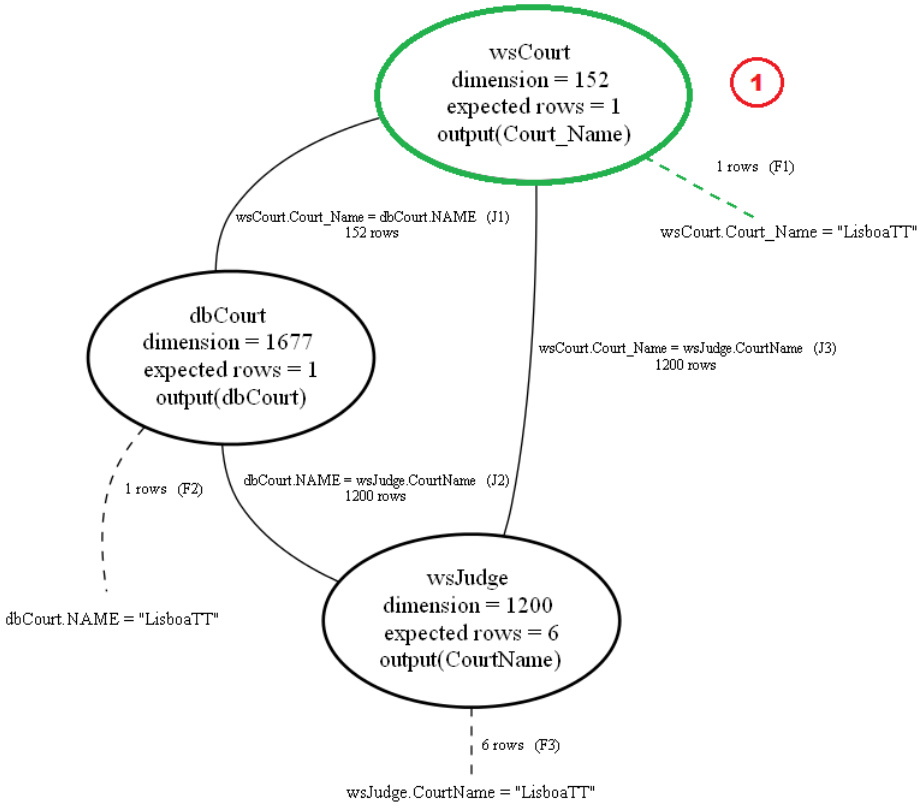


Figure 4.4: Execution algorithm: 1st step

Every time an arc is executed, the optimizer updates the queue with new costs. This part is an improvement of the algorithm presented in [GSS07]. By updating the costs of the arcs in the queue, we make our algorithm adaptive during runtime because it uses fresh information - data fetched from the entities and real joins computed - to change the remaining execution flow. While the information annotated in the graph is estimated and gathered from statistic metrics, when the optimizer actually has access to parts of data, it can precisely update the estimation for costs of future operations. These updates are performed on the direct connections of the removed arc, that is, on the joins arcs connecting the entity, or entities, involved in the executed operation.

We now show in a sequence of illustrated steps the execution produced by the optimizer for the query graph presented before. Note that during execution the amount of records retrieved from the data sources may not be equal to the estimated. Nonetheless, in this example we consider them to be the same so that we can better animate the execution of the query, shown from Figure 4.4 to Figure 4.9.

On Figure 4.4, the optimizer removes the first arc from the queue (*F1*) and executes it. In this case, no update is done on the remaining elements because the cardinality of the

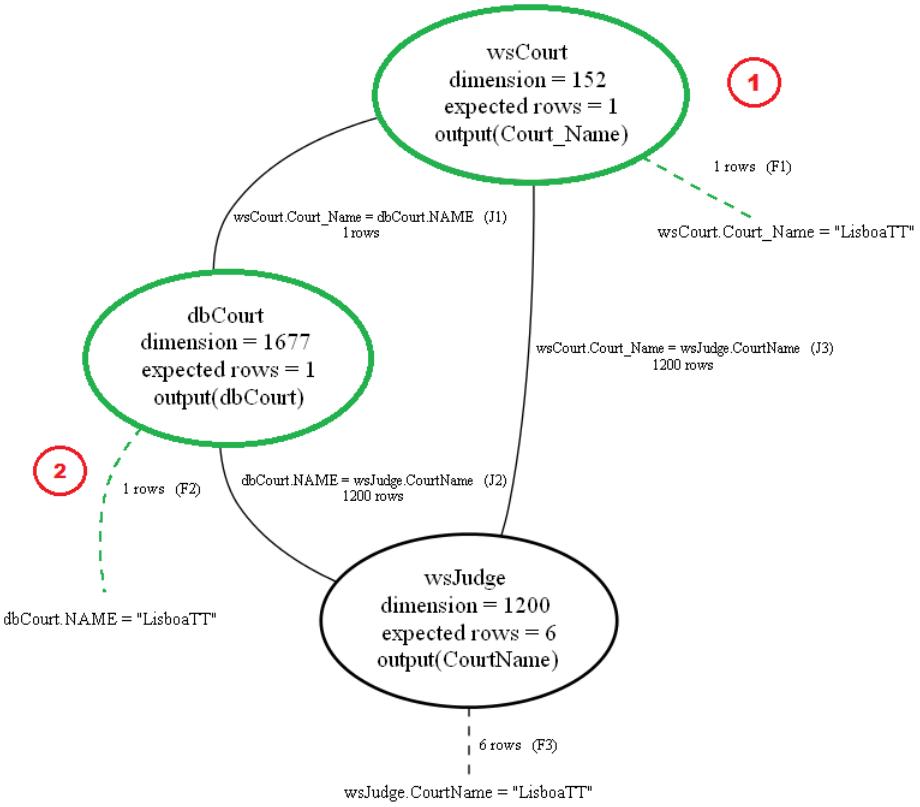


Figure 4.5: Execution algorithm: 2nd step

other operations do not change.

On Figure 4.5, the arc *F2* is removed from the queue, executed, and the cost of the join arc *J1* is updated to 1, since it is a join between two data sets of size 1. At this step, the order of the queue has already changed.

On Figure 4.6, the arc *J1* is removed and the join between the two available data sets is computed. Once again, the cardinality of the remaining operations do not change and thereby no update is performed.

On Figure 4.7, the arc *F3* is removed from the queue and executed. Since the data set resultant from the application of the filter has size 6, and the size of this data set has influence on the cardinality of the remaining join operations, their costs are updated.

On Figure 4.8, the arc *J2* is removed from the queue and the join is computed. No updates are performed.

Finally, on Figure 4.9, the last arc (*J3*) is removed from the queue and the join is computed. At this point, the queue has no more elements and thus the iterative algorithm ends. However, the resulting data set may not be yet ready to be retrieved because it may contain unnecessary columns. Thereby, the optimizer checks the output structure of the graph, which maintains the attributes specified in the query *SelectClause*, and cuts the data set, if needed. Afterwards, the data set is retrieved and query execution ends.

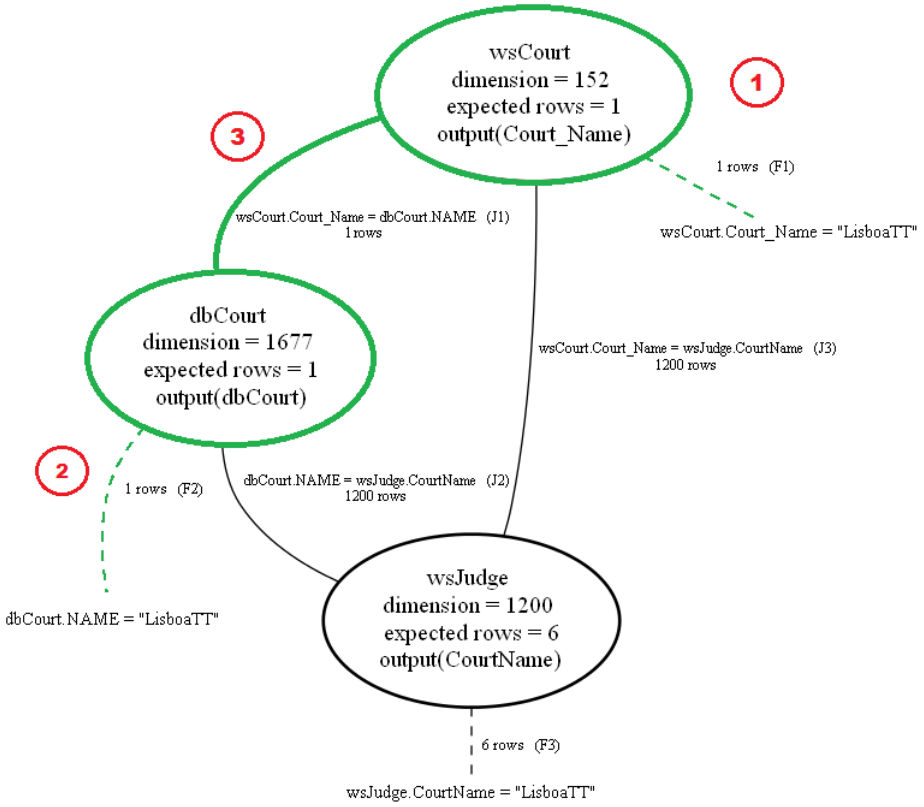


Figure 4.6: Execution algorithm: 3rd step

We do not prove that this algorithm executes the optimal query plan, but it executes an efficient one. Furthermore, the more statistics it gathers over the entities and its queries/APIs, the more precise the estimations are and therefore the more efficient the query execution is. This algorithm executes the operations with lowest costs, which in our context means executing operations that generate the minimum rows. Thus, it contributes to an efficient execution because it avoids merging large data sets, and avoids large data transfer by trying not to choose APIs or queries that retrieve all the data from the data sources. This last feature is described in the implementation section 5.4.

4.2 Model

We now present the model that supports the algorithm described in the previous section. Our model represents statistic metrics over the data sources, as well as the graph data structure for query plans. Besides, we also present some constraints over the queries we address, in order to successfully tackle the challenge presented in section 1.2.

4.2.1 Constraints

We consider the kind of queries shown in Listing 4.1. In these queries, *E1...Em* is a set of entities, which can be database tables or web services, joined through specified conditions

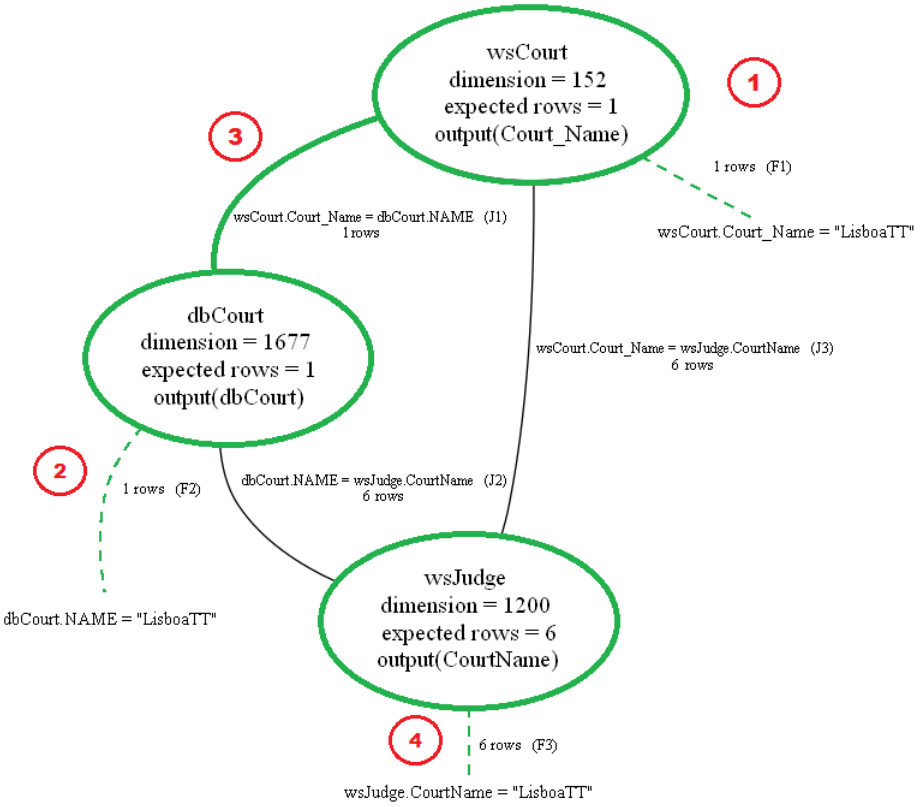


Figure 4.7: Execution algorithm: 4th step

C1...Ck. The kind of joins considered are only inner joins. Finally, *P1...Pn* are filtering operators applied to a set of attributes *A1...Am*, projected in the selection.

Listing 4.1: Type of queries over heterogeneous data sources

```
1 FROM E1 join E2 on C1 join E2 on C2, ..., join Em on Ck
2 WHERE P1(A1) and ... and Pm(Am)
3 SELECT A1, ..., An
```

In order to successfully tackle the challenge presented in section 1.2, we have auto-imposed a series of simplifications or restrictions. Some of these constraints concern the kind of queries we address, while others come from the expressiveness of the query language Linq. Every constraint concerning query operators that are not included in our model contribute to a more restrictive use of queries. Therefore, by facing these constraints developers do not have the same query expressiveness as with SQL, when querying databases.

- We do not consider join conditions other than inner joins on equalities (equi-joins)
- We do not consider queries containing sub-queries, group-by and order-by clauses
- We only address filter predicates with the operator "AND" (&&)

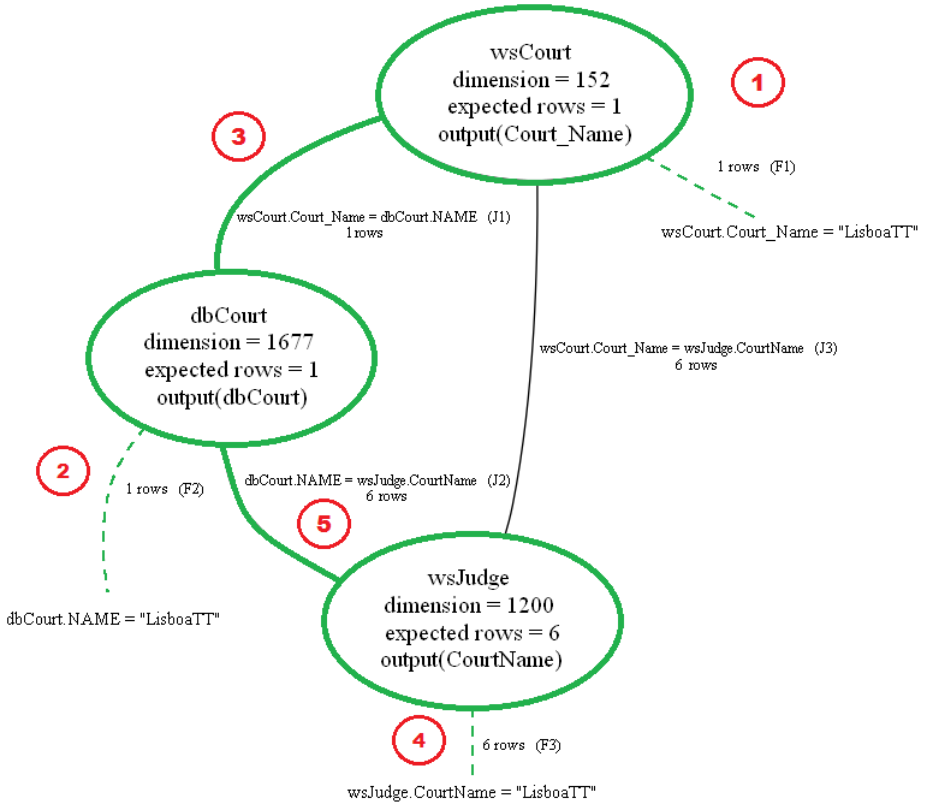


Figure 4.8: Execution algorithm: 5th step

- We do not consider the application of system or user-defined functions to results or any other part of a query
- With Linq, after writing a join in a query, we can specify one condition. Thus, for multiple join conditions we write the remaining conditions in the where clause.
- The class of web services we address are atomic web services. By this, we mean that no other attached collections are retrieved rather than the ones defined by the API output definition.
- We have only used web services receiving a single input argument
- We followed a specific name nomenclature for web services APIs. Hence, if an API is indexed via an attribute, the method fetching data from that API is named *GetByX*, where *X* is the name of the attribute, for instance: *GetByCity*, *GetByJudge*, *etc...*. Furthermore, APIs retrieving all the records from a web service have the name *GetAll*.

4.2.2 Statistics

By maintaining statistics over the data sources, specifically over entities and its columns, we aim at programming our optimizer to decide which queries and API calls are more

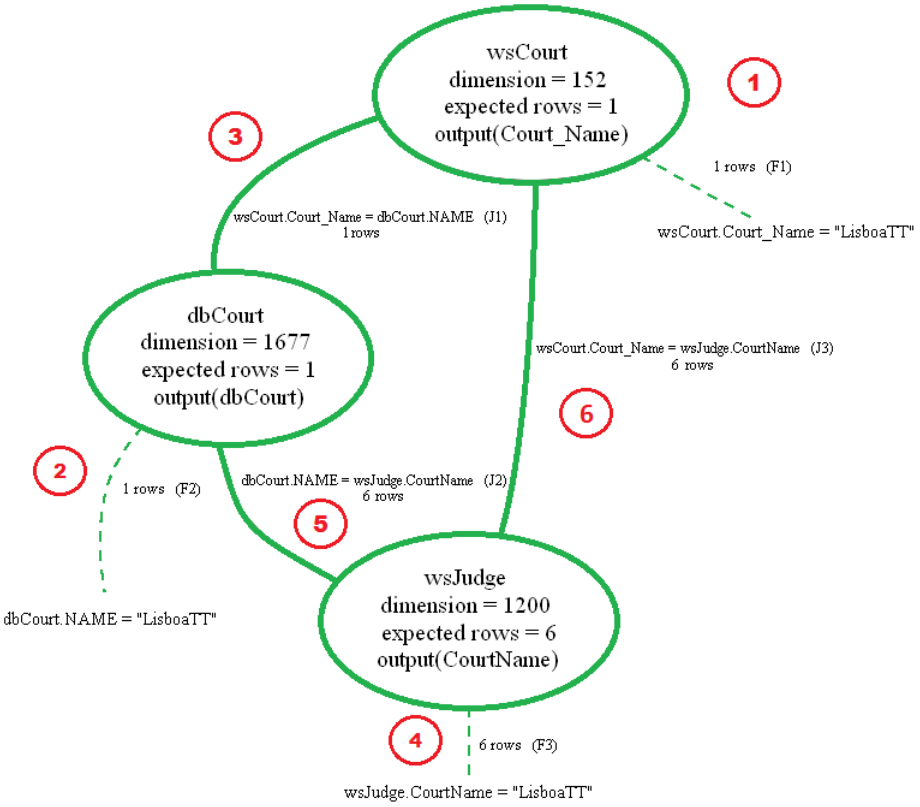


Figure 4.9: Execution algorithm: 6th step

selective or faster than others, as well as which joins are more selective than others.

Regarding to database tables and their attributes, most part of the metrics we present are already stored in database catalogs and therefore we do not store them.

As part of future work, database catalog information should be loaded to our model. Nevertheless, for web service we need to gather all these metrics. Thus, for each web service API, we keep track of:

- the average time cost for each call
- the average of rows returned by each call

These measures are used by optimizer query engine when it needs to choose the most efficient API for a web service, at an execution step. The fewer rows an API retrieves, the more selective it is and therefore if it is called sooner, the remaining query domain gets smaller. However, in certain situations, it may not be possible to understand which APIs are more selective by only checking the two averages. For instance, if the system still does not have an average of returned rows maintained over two different APIs, it may not discover which one is more selective, as shown in Table 4.1, regarding the web service *WSJudges* presented in section 2.3.

API	Avg retrieved rows	Avg time cost
GetAll(): n	?	100ms
GetByCourt(string name): n	?	60ms

Table 4.1: Statistics maintained over the APIs of *WSJudges*

API	Avg retrieved rows	Avg time cost
GetAll(): n	1000	100ms
GetByCourt(string name): n	8	60ms

Table 4.2: Statistics maintained over the APIs of *WSJudges*

Although a *GetAll()* returns all the records in the entity, a *GetByCourt(string name)* may also do the same, if all the judges have the same court. Hence, without knowing the dispersion of the data, we do not consider the *GetAll()* to be the less selective API. On the other hand, facing an opposite scenario, the optimizer query engine is able to verify that *GetByCourt* is more selective and faster and thus, it is the best choice to fetch data from the entity, in the example of Table 4.2.

To handle more precise decisions, the optimizer query engine needs more accurate statistics. Therefore, we also store the following metrics, regarding the entities and some of its columns:

- total number of rows of an entity
- percentage of distinct values for specific columns
- percentage of null values for specific columns
- uniqueness information for specific columns

We only maintain these summaries over certain columns. As referred in [Ell02], holding and maintaining these statistics for every column of all entities is space and time consuming. Thereby, we maintain these metrics for columns that can be efficiently indexed (by an API, considering a web service, or by an index, considering a database table). For the web service *WS_Courts* presented in section 2.3, the columns *Name*, *NIF* and *City* are the ones holding these summaries. The distribution of values in columns is only stored/updated when the system fetches all the data from the entity. We track the uniqueness measure for columns of web services via .NET reflection mechanisms incorporated in the programming language C#, by investigating the class containing the definition of its APIs. If an API *GetByX(...)* has a single object as its function output definition, that column is considered to be unique. On the other hand, if the function output definition indicates an array of objects, then the considered column is not unique because

```
public External_Court GetByCourt_Name(string court_name)
{
    return CallGetCourtBy_Name(court_name);
}

public External_Court[] GetByCourt_City(string court_city)
{
    return CallGetCourtsBy_City(court_city);
}
```

Figure 4.10: API investigation

Column	Unique	% distincts	% nulls
Judge	Yes	100	0
Court	No	12	0

Table 4.3: Statistics maintained over the columns of *WS_Judges*

it may retrieve a set of objects. Figure 4.10 exemplifies this scenario: attribute *Court_Name* is unique and attribute *Court_City* is not.

By holding these detailed summaries, the optimizer query engine is now capable of computing the selectivities for the columns. Table 4.3 shows an example of these metrics for the web service *WS_Judges*.

Consider now the filter *Court = "LisboaComercio"* to be applied to *WS_Judges*. Although there are two alternatives to get the data from the web service, the most efficient should be chosen:

- *GetAll()*
- *GetByCourt("LisboaComercio")*

At this step, if the optimizer query engine does not know the average number of rows available for both APIs, it computes the selectivity of the column *Court* and, if it its higher than 0, chooses the *GetByCourt("LisboaComercio")* API instead of the *GetAll()*. *GetAll()* has selectivity 0 because it retrieves all the records from the entity. Recall the concept of column selectivity presented in section 3.5, and that *WS_Judges* contains 1200 records. Thereby:

- *Court* selectivity: $\frac{144}{1200} = 0.12$

Thus, the best choice is to invoke *GetByCourt("LisboaComercio")*. Consider now this less realistic but possible scenario regarding the same web service. Besides, we want to apply two filters to *WS_Judges*:

- *Court = "LisboaComercio"*

Column	Unique	% distincts	% nulls
Judge	No	70	?
Court	No	60	?

Table 4.4: Statistics maintained over the columns of *WSJudges*

- *Judge = "John Smith"*

In this case, the column selectivies are:

- *Court* selectivity: $\frac{720}{1200} = 0.6$
- *Judge* selectivity: $\frac{840}{1200} = 0.7$

Thus, the best choice is to invoke *GetByJudge("John Smith")*. As we see, by knowing the dispersion of the values in columns, the optimizer query engine can choose the most efficient way to fetch data from entities, by finding the best ratio of distinct values. This means that less tuples will be most probably retrieved. In case of a draw, the optimizer query engine computes the best ratio of null values. The more null values in a column, the less tuples are likely to be retrieved.

To represent all these concepts, we present Figure 4.11 that shows how we structure the statistics model over the data sources. We distinguish external entities from database entities. *ExternalEntity* is an abstraction of a web service entity and it intends to hold information common to every web service. On the other hand, *DBEntity* represents a database entity/table and stores necessary metadata information. Both *DBEntity* and *ExternalEntity* offer methods to access the properties of a super class *Entity* (therefore extending it), which holds all the statistical metrics presented, over APIs/queries, entities and its columns. The concepts held in the class *Entity* are common to database entities and web services and it offers interface methods to consult and update the statistics. For a detailed description of the class *Entity*, check the appendix section A.3.3.1.

As we aim to maintain adaptive and incremental statistics, in the beginning of the query execution algorithm they are loaded, during query execution they are updated, and in the end of the execution they are permanently saved. This is achieved by storing them in the file system.

In order to connect the entities with our model, a simple step is performed. When adding a new web service to the project, we create the class containing its custom data representation and the class with the definition of its APIs. To connect the new web service with our data model, we make the APIs class extend the class *ExternalEntity*, as we show in Figure 4.12. For database entities, we do not perform this process because we do not maintain these metrics, as referred before.

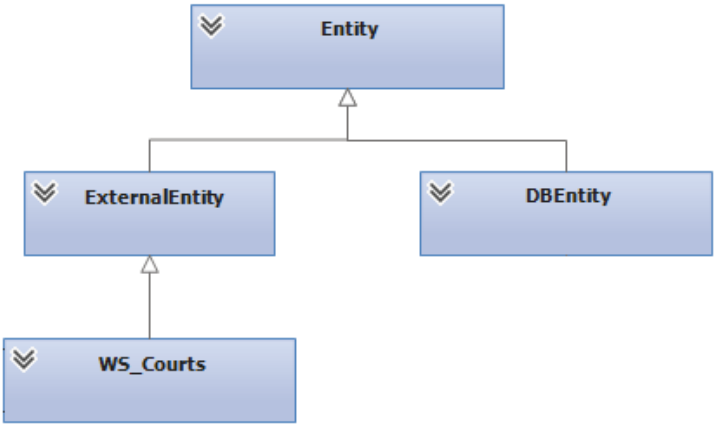


Figure 4.11: Statistics model

For details regarding to the implementation and maintenance of the statistics of our model, check section A.3.3.1 of the appendix.

4.2.3 Hints

Until now, we have been presenting calculations for specific cases where statistics are available. Nevertheless, the optimizer should be able to work even without having them, for example, the first time it is invoked. Moreover, even after some queries are executed, many APIs may have not been invoked yet, some distribution metrics over columns may not exist yet, and therefore the optimizer may not produce an efficient execution plan.

Another concept we added in our model is the knowledge of developers, which is very useful for certain situations. Typically, developers have knowledge about the structure of applications data model, as well as an idea about the size of its entities and possibly about the dispersion of the data contained over the entities. Thereby, developers can help the optimizer query engine by supplying hints regarding some metrics, contributing for its best performance. The hints we consider are:

- total number of rows of an entity
- uniqueness information in a column
- percentage of distincts in a column
- percentage of nulls in a column
- column is foreign-key for another entity's column

We chose this set because in the absence of statistics, these are the minimum hints that allow the optimizer query engine to efficiently execute queries, since it can perform a good estimation about filters and joins operations. Besides, these are simple hints usually known by a developer and they are typically stable.

```
public class WS_Courts : ExternalEntity
{
    private Courts_WS.CourtsInfo client;

    public WS_Courts(Type type, string name) : base(type, name)
    {
        base.loadStatistics();
        base.InspectAPIClass(type);

        client = new Courts_WS.CourtsInfo();
    }

    public External_Court[] GetAll()
    {
        return CallGetAllCourts();
    }

    public External_Court GetByCourt_NIF(string court_nif)
    {
        return CallGetCourtBy_NIF(court_nif);
    }
```

Figure 4.12: Connecting a web service with the data model

The foreign key hint is a metric that is not maintained as a statistic. As explained in the appendix section A.3.3.2, we store foreign key information inside the collection *foreignKeys*, in the class *Entity*. Between databases entities, foreign keys physically exist and this information can be loaded from the catalog. However, between web services or between a database and a web service, the concept of foreign key does not directly exist like in a database, although it can be simulated as such. Storing foreign key information on columns is important to estimate the output cardinality of joins, as presented in section 3.5. This metric allows the optimizer query engine to exactly compute the number of resulting rows from a join, providing precise information.

Developer hints are supplied as annotations inside the classes representing the custom data arising from the entities, as presented in Figure 4.13.

The total number of rows of an entity can be supplied via an *EntityHint*, above the name of the class, while the hints regarding columns can be supplied through *ColumnHint*, placed above the definition of the properties. Finally, the foreign key information can be also given inside a *ColumnHint*, as presented in Figure 4.14, which is the continuation of the *ColumnHint* given for the property *Court_City*. In a foreign key hint, the developer needs to supply the entity to which the column is referring to, and the referenced column.

Statistics and hints are used by the optimizer query engine to populate information on query graphs. Then, query graphs and their annotated information are used during query execution. The optimizer query engine gives priority to statistics over hints. In other words, if a developer supplies a hint and that information is already available as

```
[EntityHint(totalRows = 150)]
public class External_Court
{

    [ColumnHint(unique = true, distincts_ratio = 100, nulls_ratio = 0)]
    public string Court_Name { get; set; }

    [ColumnHint(unique = true, distincts_ratio = 100, nulls_ratio = 0)]
    public string Court_NIF { get; set; }

    [ColumnHint(unique = false, distincts_ratio = 35, nulls_ratio = 10,
    public string Court_City { get; set; }
```

Figure 4.13: Information supplied by developers

```
tio = 10, foreign_key_to = typeof(Cities_WS.Cities_WS), references_attribute = "City")]
```

Figure 4.14: Information supplied by developers

a statistics, the optimizer query engine ignores such hint. Nonetheless, by supplying precise hints, developers ensure that queries can continue to be efficiently executed, even if something happens with the statistics files, or if they are reset.

4.2.4 Query Plan Graph

As presented, we represent our query plans in graph structures so we can implement the query execution algorithm. Thus, we need a support structure for it, which allow us to iterate over the structure of a query and incrementally create nodes and arcs with related information. Figure 4.15 shows the data structure for our query plan graphs.

The most two basic structures in our query graphs are the node and the arc. A node represents an entity, which can be either a database table or a web service, while an arc either connects two nodes (representing thereby a join), or connects a node to a filter being applied to that node (a where clause predicate). All the common information to both entities is represented in a super class *Node*, which is inherited by a *DBNode* and a *WSNode*. These two classes directly identify the type of the node, even though they do not contain their own specific attributes. All the necessary information is stored in the super class *Node*:

- *joins*: a collection containing all the joins (arcs) where the node is involved
- *filters*: a collection containing all the filters (arcs) applied to the node
- *apis*: a collection containing all the available APIs to be invoked for the entity. If the node is a *WSNode* they are API calls, otherwise they are queries.
- *outputStructure*: the minimum set of attributes of the entity required to be stored for the execution of the query

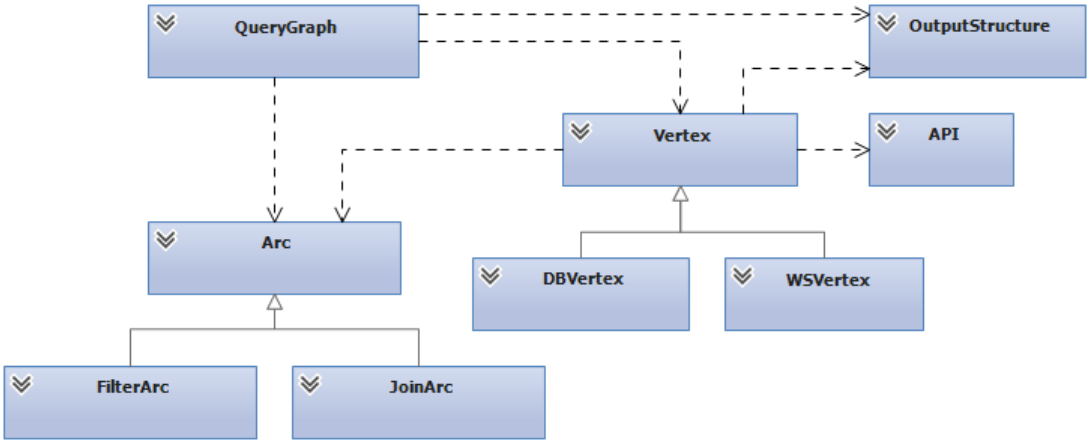


Figure 4.15: Query plan graph data structure

- *dataSourceID*: the identifier associated to the data source. For instance, if two database tables belong to the same database, they have the same identifier, while two web services have different identifiers.
- *totalRows*: total no. of rows of the entity
- *expectedRows*: expected no. of rows resulting after the application of all filters
- *name*: name given to the entity in the query
- *output*: collection that may contain data fetched from the entity, populated during query execution

There are two kinds of arcs in the graph, joins and filters. A join is represented as a *JoinArc* and it connects two nodes, containing a reference to both plus a join condition. On the other hand, a *FilterArc* connects a node to a filter and therefore it contains a reference to a node and an expression. Both types of arcs have two attributes in common: a condition expression and the expected number of rows resulting from that operation. Thus, these two attributes are represented in the super class *Arc*.

A call or query for an entity is represented in the class *API*. There, we store its expected number of rows and its expected time cost, as well as the name of the API/query, along with the input parameter. Since we just use web services receiving a single argument, this input parameter is an object instead of a list of objects. For queries, we simply represent equalities like *Entity.A = "X"*, which are translated for SQL queries like *SELECT * FROM ENTITY WHERE A = 'X'* during the graph execution. Finally, the graph is represented in the class *QueryGraph* and it contains:

- *filters*: a collection with all the filters
- *joins*: a collection with all the joins

- *nodes*: a collection with all the nodes
- *outputStructure*: the output structure specified in the query *SelectClause*
- *rows*: no. of rows retrieved
- *cost*: time cost of its execution

5

Implementation

In this chapter we describe the main parts of the implementation process of our solution. We implemented an optimizer query engine that executes the kind of queries we address, following the query execution algorithm presented in chapter 4.1. The optimizer query engine is supported by the model of metrics gathered over calls/queries, entities and its columns, presented in section 4.2.

In order to be able to develop a query execution algorithm for Linq queries, we used the framework Re-Linq and the programming language C# to develop a query provider, and the query language Linq to write our queries. This project was implemented with the help of Visual Studio 2010, a Microsoft tool to build and develop projects and applications.

A detailed description about how to build Re-Linq sources and more context about the tool we developed, is presented in the appendix chapter A.2.

5.1 Querying data sources

In this section we show we query databases and web services using Re-Linq and C#. While to query a database we use the Linq provider *Linq-To-SQL*, to query a web service we still have to develop some code.

5.1.1 Executing a query in the database

To execute a query against the database with Re-Linq, we need to have an SQL command. Since by using Re-Linq we work with *QueryModels* as the representation for a query, we need a way to translate them into SQL commands, so we can supply them to *Linq-To-SQL*

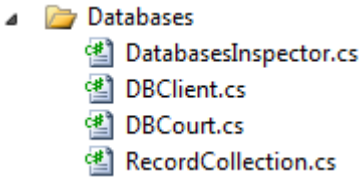


Figure 5.1: Database classes

to execute them against the database.

Re-Linq has a back-end tool that accepts a well-formed *QueryModel*, interprets it, and retrieves the equivalent SQL command with the related parameters. This tool is contained within the library *Remotion.Linq.SqlBackend*. Thereby, we can use this back-end tool to automatically generate an SQL command and supply it to *Linq-To-SQL*, which retrieves the database records.

The other option opposing this is to manually parse a *QueryModel* and generate a string with an equivalent SQL command. Afterwards, such SQL string can be supplied to *Linq-To-SQL* that executes it against the database.

When executing queries against databases, we store the results retrieved in custom entities representing the structure of the data of the entities. Those are the classes where developers may supply the hints and they are placed inside the package *Databases*, presented in Figure 5.1 (classes *DBClient* and *DBCourt* for our scenario). The class *RecordCollection* is used for creating a collection of records that is stored in temporary tables, a feature explained in the implementation section, used for certain kind of joins. *DatabasesInspector* is a mediator object holding a collection of database entities existing in a query and is responsible for supplying a specified database entity object to the optimizer or to the graph generator, so they can update or check its statistics.

5.1.2 Executing a web service API

In order to call a web service API with our tool, we first have to generate some C# classes. For each web reference added to the project (recall section 3.6.1), we create a class containing the actual execution of its APIs calls and related result handling. Thus, for a web service of judges we generate a class named *WS_Judges.cs*. Furthermore, we also create a custom entity class representing the external entity data structure, to store its data and hold possible hints given by developers. For instance, we add a class named *External_Judge.cs* for the web service of judges. For each custom entity class added, it is also needed to add the respective hints access class to the package *Hints*, as described in section 4.2.3. As an example, for the *External_Judge.cs* class, a class *External_Judge_Hints.cs* is therefore needed.

All these classes (except the hints access class) are located inside the package *WebServices*, as shown in Figure 5.2. The class *WebServicesInspector* is a mediator object needed

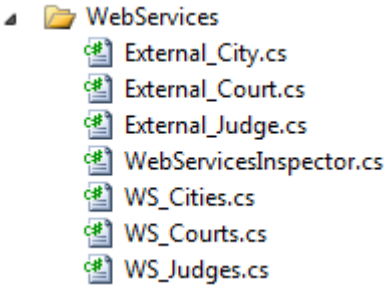


Figure 5.2: Web service classes

for *GraphGenerator* and for the optimizer query engine, with the same purpose as *DatabasesInspector*, holding a collection of web service entities existing in a query and is responsible for supplying a specified web service entity object, so they can update or check its statistics.

Each custom entity class contains the fields existing in the entity, so when the results of a query or an API call are returned, new instances of these classes are created, storing retrieved data. For an illustrative example of the invocation of the API *GetByCourt_City* for the web service *WS_Courts*, check Figure A.14 located in the appendix.

5.2 Execution flow

The query execution algorithm is implemented in *ExecuteCollection* method. We start by filtering some useful information from a *QueryModel*, by iterating it and keep in different collections the database and web service entities specified in the query. With these collections, we are able to easily detect whether a query is selecting data from databases, from web services, or from both data sources. Besides, they are necessary to to initialize two inspector objects: a *DatabasesInspector* and a *WebServicesInspector*. These two objects are placed inside the packages *Databases* and *WebServices*, respectively, and they save some metadata information plus the representation of the entities (*DBEntity* and *ExternalEntity*) used in the specified query, so the optimizer query engine can access their statistics and call their APIs and execute queries, during query execution. Furthermore, these objects are also responsible for loading their respective entities statistics, if available, and for serializing the statistics generated during the query execution algorithm, assuming that the query executes properly.

Finally, we also store in different auxiliary structures, the attributes of database and web service entities specified in the query *SelectClause*. This is achieved through visitor algorithms located inside the package *QueryClausesTransformer*. For further information regarding visitor algorithms in the context of Re-Linq and Linq expressions, check the references [Gie, Fab]. All these auxiliary collections are useful for the parsing algorithm and for the optimizer query engine.

For a given Linq query, our algorithm captures whether the query is dealing only

```
// Database query: send to DBMS
if (this.WS_DataSources.Count == 0)
    databaseResults = dbInspector.ExecuteQuery(queryModel);
else // Web service, or mixed query
```

Figure 5.3: Database queries detection

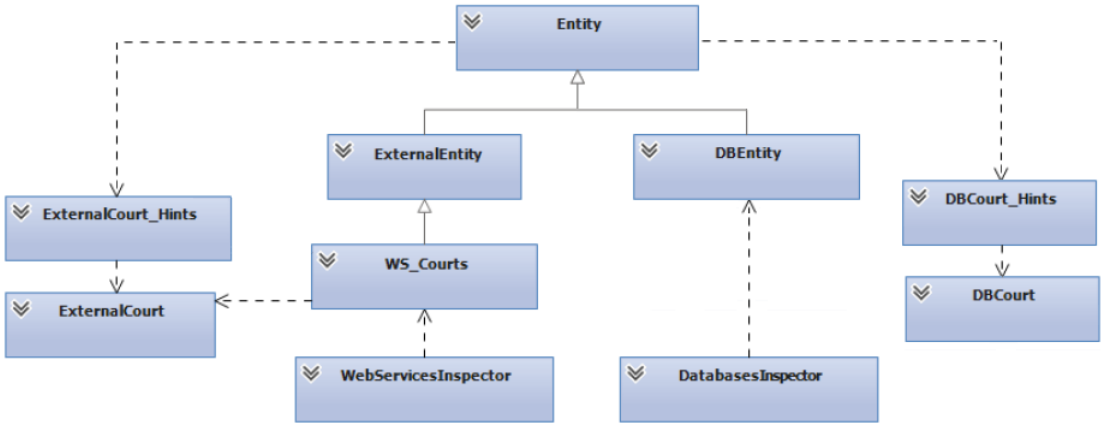


Figure 5.4: Final model structure

with database entities, web services, or both data sources. Hence, for queries concerning only data from database entities, our parsing algorithm and the optimizer query engine need not to be invoked, since queries can be directly sent to the database. Possible optimizations regarding such queries are performed by the database management system.

To verify whether we are dealing with this kind of queries, we check if the collection holding the number of web service data sources is empty. Figure 5.3 shows the part of the code implementing this feature.

Since the query plan graph need statistical information, before invoking the parsing algorithm we load the web service entities statistics to their respective class collections. This is achieved by initializing the inspectors classes. For database entities, the initialization of *DatabasesInspector* would load the relevant information from the database catalog to the respective entities, a feature that we do not implement. Now that we presented the intent of the inspector objects, the final architecture of our data model is as shown in Figure 5.4.

5.3 Parsing a *QueryModel*

Our parsing algorithm, from here on *GraphGenerator*, generates a query plan graph given:

- a Re-Linq *QueryModel*
- a database related select clause

- a web service related select clause
- a *DatabasesInspector*
- a *WebServicesInspector*

The graphs generated follow the structure explained in section 4.2.4. To begin, *GraphGenerator* checks the *MainFromClause* of a *QueryModel* and generates the corresponding entity node, whether the data source is a web service or a database entity. When creating the node, all available information is populated in its class, such as its name, the total number of rows of the entity, as well as other metadata information. In the end, the new node is added to the graph.

On the second step, *GraphGenerator* checks the *BodyClauses*. For each *JoinClause* detected, it generates a new node regarding the new entity appearing in the expression, adds it to the graph, and connects both entities through a new *JoinArc*. When creating a *JoinArc*, *GraphGenerator* defines the involved nodes and the condition representing the join expression. When *GraphGenerator* detects a *WhereClause*, no more *JoinClauses* can appear and thereby it iterates the contents of the *WhereClause*, creating a *FilterArc* for each expression and adding it to the graph. For each filter discovered, *GraphGenerator* connects it to the respective entity. Figure A.15 located in the appendix shows this part of the code. Due to the extension of the code, we hide some parts by replacing them with *(...)*. Nevertheless, they regard not important details like input arguments passed to other functions.

With the main structure of the graph built, we now generate new possible connections, discovered through the commutativity of certain operations, and populate statistical information and cost metrics in its elements. For a detailed description about this process, consult the appendix section A.3.1.

5.4 Optimizer query engine

Our optimizer query engine automates the process of data integration of databases and web services written with Linq queries. By choosing the most efficient execution flow that computes the minimum number of records at each execution step, and by invoking the most efficient API calls, queries are efficiently executed. Besides, the system is adaptive over the time because it maintains moving averages of times and rows retrieved by APIs, and tracks changes on the distribution of data in the entities. Due to this reason, the algorithm will execute queries efficiently even when data changes with time.

Therefore, we automate and solve a process that is usually done by developers in companies, when integrating data from database tables and web services: the manual implementation of a data retrieval algorithm and the insurance of its efficiency.

```
QueryExecutor executor = new QueryExecutor(wsInspector, dbInspector, query_graph);
executionResults = executor.ExecuteGraph();
```

Figure 5.5: Invoking the optimizer query engine

5.4.1 Execution algorithm

Recall the description of the execution algorithm in section 4.1. Since the algorithm is supported by a query graph, our optimizer query engine receives a query graph on its initialization. Moreover, it also receives a *DatabasesInspector* and a *WebServicesInspector*, in order to be able to consult the statistics and hints maintained over the data sources. This is shown in Figure 5.5.

As described before, the optimizer query engine executes a query graph by following the arcs with minimum costs at each step. Hence, upon its initialization, it generates a priority queue of arcs ordered by minimum costs. When two arcs have the same cost, the ordering function prioritizes filters to joins, as well as filters applied on database entities. Such function is implemented in the class *Arc* for every arc, by implementing the interface *IComparable* and therefore implementing the method *CompareTo*. Once the queue is built, the optimizer query engine is ready to execute the graph.

The execution of the graph starts with the invocation of the method *ExecuteGraph()*, as showed in Figure 5.5. Figure A.25, located in the appendix, shows the main recursive function of the optimizer query engine.

During query execution, the optimizer query engine stores visited data in a system variable named *visitedData*, so it can access the data sets extracted from the data sources, either to build indexes or to any other necessary operation. This collection is detailed ahead.

5.4.2 Execution of filters

We begin by explain how the optimizer query engine executes filters over web services. The first step is to find out which is the best API to invoke. This feature is implemented as a minimum time cost calculation on the set of available APIs. However, for an API using data that is already available, the optimizer query engine does not only look to the time cost of the API, since it may have to be invoked several times, according to the size of the available data set. Hence, it multiplies the length of the available data set with the time cost of that API, which gives the final cost. As an example of a scenario like this, consider the available set of cities *{Lisbon, Porto, Sintra}* and a join between cities and courts. Since there is an API *GetByCourt_City* for the web service of courts, we can produce the join by invoking three times that API (one for each city available).

Although the minimum cost is the base computation for finding the best API, we made this computation a little smarter. Consider the web service of courts with a cost of 300ms for the *GetAll* API, and a cost of 101ms for the API *GetByCourt_City*. Consider also

the set of cities {*Lisbon, Porto, Sintra*} and the join being issued between cities and courts.

Following the computation described, the API *GetAll* would be the best since 300ms is smaller than 303ms. However, by invoking *GetAll*, a later memory join has to be performed to compute the join of both collections, while calling three times *GetByCourt_City* already produces the join result. Therefore, the optimizer query engine maintains a system variable representing an extra delay that is added for APIs that do not produce the final join result, during this minimum cost computation, so the optimizer query engine can improve query execution efficiency. This variable is called *DIFFERENCE_ALLOWED_FOR_API_CHANGE*.

After the API is found, the optimizer query engine accesses the web service entity through the *WebServicesInspector* and invokes it. Once the data is fetched, the optimizer query engine applies all the available filters to the data set and the filter execution is terminated.

Executing filters over database entities is easier. Once this case is detected, the optimizer query engine takes all the filters being applied to the entity and invokes an utility class *QueryGenerationHelper* built by us, that generates an SQL query. Since we do not have a *QueryModel*, we cannot use the automatic tool of Re-Linq that generates an SQL command and thereby we implemented this SQL command generation mechanism ourselves. To execute the SQL query, the optimizer query engine invokes *DatabasesInspector* that executes it and retrieves the results from the database, inside of classes representing the custom entities, for example in classes of *DBCourt*, as shown in Figure 5.1.

5.4.3 Memory joins

Before explaining how the optimizer query engine computes the joins of data sources, we describe how memory joins are computed. When facing joins of collections in memory, a fast strategy has to be implemented because these operations are time-consuming. Therefore, we chose to use the strategy of indexed nested loop joins (section 3.5) to implement memory joins of data sets.

Every time the optimizer query engine fetches data from an entity, if the queue still has arcs to remove, an index is built on that collection. By following this strategy, we ensure that futures joins using this entity can be executed as indexed nested loop joins, since an index already exists. In certain execution flows, it may happen that both entities already have indexes built on their data sets and therefore the join is performed by only iterating the smallest collection. For cases when a join is to be executed and none of the entities were already address before, their data is not available. Thus, the optimizer query engine builds an index over the data set of the first entity fetched, so it can afterwards execute the merge efficiently, when it gets the data from the second entity. A more precise approach could have been implemented, since for joins where collections are small, a naive nested loop join may perform as fast as an indexed nested loop join and thereby there is no need to lose time building an index.

```
private readonly double DIFFERENCE_ALLOWED_FOR_API_CHANGE = 3.0;

private WebServicesInspector ws_inspector;
private DatabasesInspector db_inspector;
private QueryGraph graph;
private SortedList<Arc, int> queue;

// auxiliary variables
private Dictionary<string, Arc> removedJoinArcs;
private Dictionary<string, string> entitiesVertexNames;

// indexes builts over data sets
private Dictionary<string, Dictionary<string, Hashtable>> indexes;

// Data fetched from the data sources and joined during execution.
// Each group of vertexes is a collection of tuples.
// A tuple is a Dictionary<string, object> because it may have data
// from several entities on its columns.
private Dictionary<List<Vertex>, List<Dictionary<string, object>>> visitedData;
```

Figure 5.6: Optimizer query engine variables

When an index is built over a collection of records, the optimizer query engine detects which attributes should be indexed. These attributes are the ones being used in join arcs that were not yet executed (they still exist in the execution queue). When a join between two collections is computed, resulting therefore a new collection, no index is built on the new collection until another join with such collection is to be executed. At that moment, the optimizer query engine decides on which collection it should build the new index.

Building indexes is space-consuming because it implies maintaining in memory hash tables that may contain hundreds or thousands of attributes. Hence, every time an index is not necessary, that memory space is cleared.

We present in Figure 5.6 the system variables defined and maintained by our optimizer query engine, supporting all this algorithm.

5.4.4 Execution of joins

Before explaining how join arcs are executed, we describe how data computed during execution is stored in memory. Recall that data sets arising from the data sources are stored in the collection *visitedData*. This collection holds data fetched from entities and join results produced at each execution step.

When performing a join between two entities that were already addressed before (a *FilterArc* was executed for each entity, for instance), *visitedData* contains a data set for each one of them, stored in a different index. The join is then computed and the final result stored, associated to both entities. To achieve this functionality, *visitedData* is implemented as a *Dictionary*, where its key is a list of nodes representing the entities to which the data set, stored in its value, is associated. Although a more efficient data structure

may be used for this purpose, this representation does not produce performance issues since these query graphs have a small amount of nodes and thereby search or update operations are not costly.

When the optimizer query engine removes a join arc from the queue, it faces three different scenarios. The first scenario occurs when both entities that are being merged have already been address before and therefore data is already in memory. Thus, the optimizer query engine is able to merge the data sets contained inside *visitedData*. If both entities belong to the same list in *visitedData*, meaning that these entities were already merged through other join conditions, the optimizer query engine iterates the data set and applies the join condition, updating the result collection in *visitedData*. On the other hand, if the data sets are not stored together, the optimizer query engine verifies which indexes are available over the data sets, thereby finding the most efficient way to perform the join. If no index is available, the optimizer query engine builds an index over the largest data set and the join is computed.

The second scenario happens when only part of the data is available and, in order to produce the join, the optimizer query engine still needs to fetch data from the other data source. Depending on which data source still needs to be addressed, a different strategy is followed:

1. The new data arises from a database: the optimizer query engine uses the available data set to build a specific query to send to the database. Three kinds of SQL queries may be built, depending on the size of the data available. If the available data set has a single record, the optimizer query engine builds a query of the type *SELECT .. FROM TABLE WHERE DB_ATTR = X*, where *X* is the value of the attribute available in the data set. The attributes considered are the ones specified in the join condition. On the other hand, if the data set has between 1 and 10 records, a query of the type *SELECT .. FROM TABLE WHERE DB_ATTR IN (LIST)* is built, where *LIST* contains a set of values from the available data set. Finally, if the data set contains more than 10 records, a join query is built between the database table and a temporary table containing the data from the available set. Such temporary table is built dynamically in runtime through Linq-To-SQL, with the help of class *RecordCollection*, placed inside the package *Databases*. This class holds an iterator of SQL data records that are supplied to Linq-To-SQL to generate the temporary table. For details about this feature, check [sitb]. The reason for building these last two different kind of queries arises from their execution efficiency, as referred in [sitc, SBB03].

2. The new data arises from a web service: the optimizer query engine finds the best API to invoke for the web service. If such API does not involve the available data, the optimizer query engine executes it, otherwise a more careful strategy is followed. Considering the set of cities *{Lisbon, Porto, Sintra}* and the invocation of an API *GetByCourt_City*, the optimizer query engine invokes three times the API, each

time with a different city, joining the results at each step. In these scenarios, the optimizer query engine takes the distinct values from the data set, since they may have repeated values, therefore avoiding repeated API calls.

For any of the last two approaches, after data is fetched from the new entity, the memory join is computed, if needed. While for a join between a database entity and a temporary table the result produced is the final join result, for other situations this may no be true. For instance, considering a large available data set and the invocation of a *GetAll* API for a web service, the join still has to be computed in memory, since the strategy chosen did not produce it automatically.

Finally, the last scenario concerns situations when no data from both data sources is available. Facing these scenarios, the optimizer query engine chooses a starting node to fetch data from, following a specific criteria. If there is a foreign key hint over the attributes referenced in the join condition, the optimizer query engine establishes as starting entity, the one being referenced by the foreign key. When no foreign key information over the attributes specified in the join condition is available, or for join conditions with multiple expressions, the optimizer query engine chooses to fetch data from the smallest entity.

Once the starting node is established, the optimizer query engine fetches the data from it, applies possible available filters, and builds an index over the resulting data. As for the remaining execution, it proceeds as explained in the previous case, where there is an available data set and a data source to fetch data from.

5.4.5 Merging database nodes

A situation that is not addressed in this work is the possibility of changing the order of certain operations in the queue, therefore trying to execute several operations in a data source at once, such as joins between database entities spread along the execution queue. By doing this, the optimizer query engine may avoid executing several queries at a database during the execution, by executing one or more at once, such as a join between one or more entities. Nevertheless, the models presented for queries and query graphs sustain this possibility. To implement this feature, only a different way of handling the execution queue needs to be implemented. The existence of the data source identifier *dataSourceID* (Figure 4.2.4) in each node is needed for these optimizations because only operations over entities belonging to the same data source could be joined.

Although we did not implement these features, we do implement an optimization in this version of the optimizer query engine, since it does not imply changing the execution queue order. Hence, for each join arc removed from the queue, the optimizer query engine detects if it merges two database entities, which data was not fetched before. In positive case, the optimizer query engine builds a unique query joining both tables and sends it to the database, retaining the join result afterwards.

Our tool automates and optimizes the development of algorithms to merge data from databases and web services that generate data for some goal. With our solution, developers need no more to manually change the algorithms to achieve such goal, since the optimizer query engine does it for itself, executing and optimizing queries over those data sources. As we showed previously, the system is not closed because it is possible to help/feed the optimizer query engine, by supplying hints. Presented the algorithm for executing these queries, we now reveal the results it achieved.

6

Results and validation

Our optimizer query engine needs to be tested and validated, so we can be sure of its real value. In order to achieve a valid set of tests and truly validate our solution, we developed an interview and applied it to five developers of *OutSystems* teams. We chose to follow an iterative exercise, where in the first version we realized what was not clear, or was incomplete, so in the following interviews we could get proper validated results.

Hence, we built an exercise where we presented a data model containing some entities that we used during our project (for development and testing purposes) and many details regarding them, followed by three exercises. Each exercise aims on building an execution algorithm for a different query, since these scenarios are common in the company and they may be developed every day by developers, to build parts of applications, such as web pages, where these data integration are needed.

For the development of each exercise, every developer produced a Linq version of the algorithm that produces the integration desired, which we implemented separately from our solution and we tested its execution. Afterwards, we wrote the equivalent Linq query and we ran it with our solution, comparing the development effort and the time efficiency of both versions. When both solutions were ran, the optimizer query engine had already been ran around three times (for some different queries), so it had already gathered some times and metrics, thereby being able to produce an efficient query plan.

Figure 6.1 shows the scenario for the exercise. The model contains 4 entities: 2 web services and 2 databases. For each entity, we show an example of populated data, the dimension of the entities and several column information as well. Finally, for web services, we show which APIs are available for use.

In this first version, we did not reveal the average of times maintained by the system

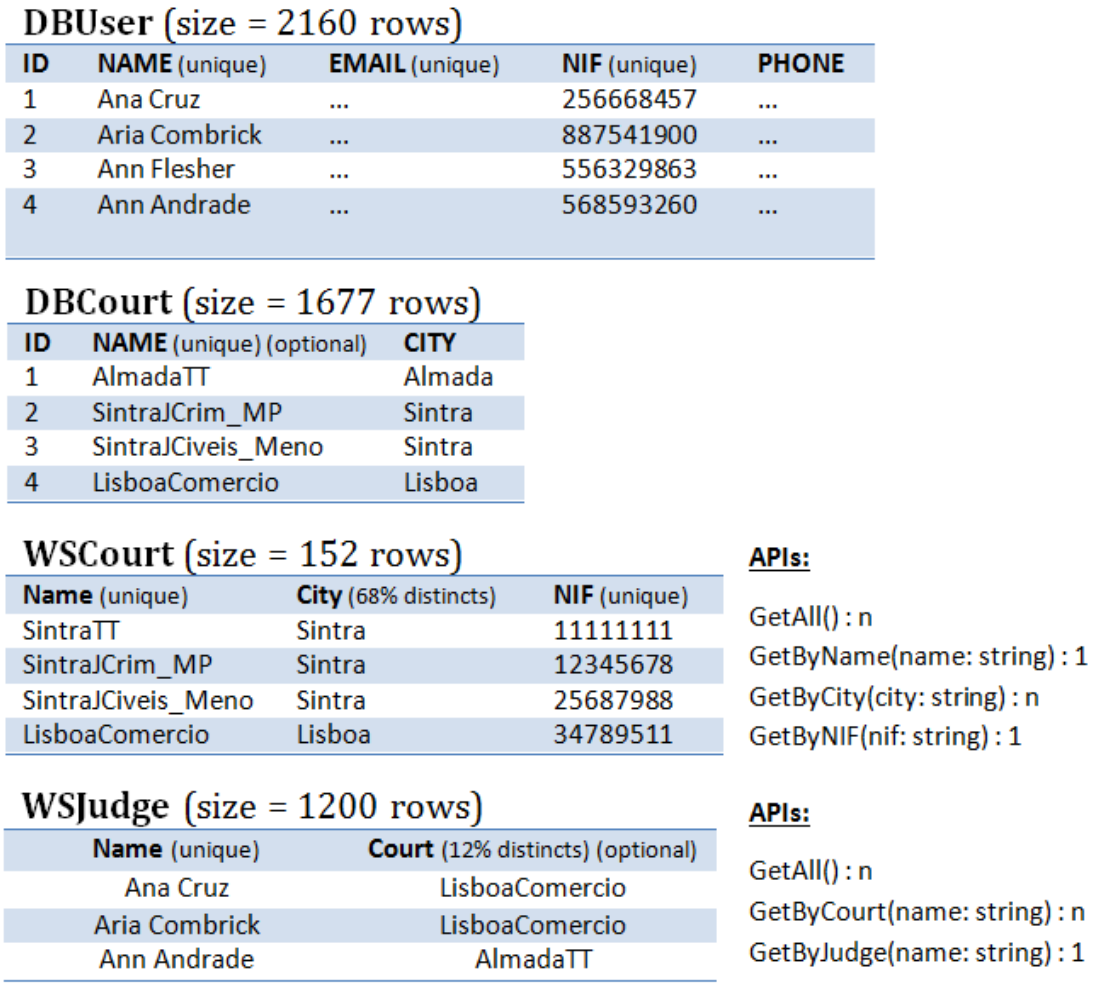


Figure 6.1: Exercise scenario, first version

over the APIs of web services because we wanted to find out the difference it would produce for the exercise. Figures 6.2, 6.3 and 6.4 reveal the three exercises proposed to the developers. For each exercise, we showed below the description which entities need to be joined to perform the integration.

Table 6.5 presents the results achieved by the algorithms developed by the first developer. For the first query, the developer produced an algorithm that issued too many web service calls to *WSCourt*, which could be avoided by fetching once all the data. While the optimizer query engine could execute the query in 0.624 seconds, the algorithm of developer 1 took almost the double. This situation would have been avoided if the developer had had access to the average times maintained over the APIs.

For the second query, the developer produced nearly the same algorithm as the optimizer query engine and it achieved almost the same performance. This may happen in scenarios where the query is simple and small data sets are involved. In this case, the

Query 1: Find all the judges working in courts of Lisbon.

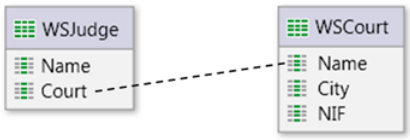


Figure 6.2: Exercise 1

Query 2: Find all the NIFs of the courts of Lisbon that exist both in "DBCourt" and in the "WSCourt".

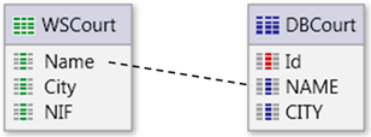


Figure 6.3: Exercise 2

manual algorithm was a little faster, even though a less efficient execution was produced, comparing with the optimizer query engine. The optimizer query engine performs many verifications during query execution (adaptive computations, consulting statistics, adapting execution queue, etc...), reason why we think that, for these simple query scenarios, the times achieved have nearly the same values.

Finally, for the third and most complicated query, the results are really expressive. When the amount of entities in a query rise, more possibilities of execution arise and the harder is for a developer to produce the most efficient algorithm. Furthermore, it became even more difficult without knowing the times maintained over the APIs. The poor performance achieved by the developer is justified by the not appropriate execution flow chosen and to the memory joins produced, which were computed as normal nested loops, without indexing strategies.

Considering the overall performance of the algorithms produced by the first developer, we decided to broaden the knowledge of the developers by supplying them the times maintained over the APIs of web services. We did not reveal the average of retrieved rows maintained over the APIs because for these exercises they did not make any difference. At this point, we tried to place the developer at the same knowledge level of the optimizer query engine, so we can fairly compare and show the efficiency of their solutions for these exercises. Figure 6.6 shows the update done to the exercise.

Thus, we present in Table 6.7 the results obtained with the remaining four developers, for all queries.

As we can see, the general efficiency of the algorithms produced by the remaining

Query 3: Find the emails of the judges working in courts of "WSCourt".

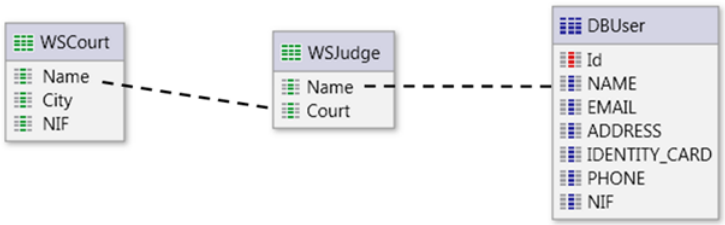


Figure 6.4: Exercise 3

Developer 1	Developer's algorithm performance	Optimizer performance
Query 1	988 ms	624 ms
Query 2	516 ms	554 ms
Query 3	21.910 ms	703 ms

Figure 6.5: Results of interview to developer 1

developers improved with more knowledge supplied, more specifically with the average of times over the APIs revealed. With such information, it became easier to understand the best execution flow for the first two queries and nearly all the developers developed a good solution, with nearly the same performance of our optimizer query engine, or even a little better. We believe the reason for the performances differences with these two scenarios to be the same as we explained before.

However, for the third query, larger performance differences occurred. Only one of the developers, not surprisingly the one with more experience, could develop nearly the same algorithm as the optimizer query engine, reason why it achieved a really close execution time. Basically, he discovered the best execution flow to follow, fetching only necessary data. As for memory joins, he used indexed nested loop strategies instead of normal nested loops. This developer has a strong knowledge about data flow and thereby he could achieve a really close performance to our optimizer query engine.

We did not show the execution plans produced by developers and by the optimizer query engine due to the large amount of space needed for that, considering all the developers and the exercises. However, we now reveal two examples of a query plan produced by one of the developers for the first and the third query. Afterwards, we compare the development effort of their implementation with our solution. Figure 6.8 reveals an example of an algorithm produced for the first query, revealing as well how it could be implemented with our solution. The same scenario is shown for the last, and more complicated query, this time showing the algorithm developed by the most experienced developer, in Figures 6.9 and 6.10.

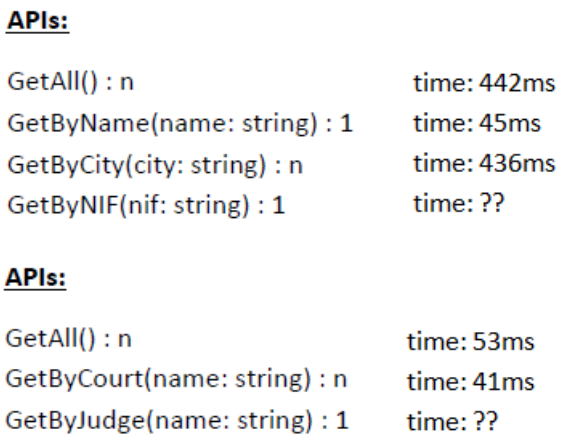


Figure 6.6: Exercise scenario updated

For these exercises, we considered integrations with two and three entities that may simulate a perfectly real scenario, considering the context we presented. Nonetheless, the amount of data we placed inside the entities was not large and in real integration scenarios in large applications, not only two or three entities are typically integrated. Therefore, our optimizer query engine shows itself useful, since no development effort is needed and the optimizer query engine is adaptive to changes on entities, their data, and queries/APIs. When queries rise more complicated, developers have to lose time manually changing their algorithms because they started to have a poor performance. With our solution, such hard work is not needed any more and we proved that our algorithms are efficient as well.

Developer 2	Developer's algorithm performance	Optimizer performance
Query 1	570 ms	624 ms
Query 2	510 ms	554 ms
Query 3	4.560 ms	703 ms
Developer 3	**Developer's algorithm performance**	**Optimizer performance**
Query 1	1000 ms	624 ms
Query 2	500 ms	554 ms
Query 3	7000 ms	703 ms
Developer 4	**Developer's algorithm performance**	**Optimizer performance**
Query 1	560 ms	624 ms
Query 2	500 ms	554 ms
Query 3	1.259 ms	703 ms
Developer 5	**Developer's algorithm performance**	**Optimizer performance**
Query 1	978 ms	624 ms
Query 2	510 ms	554 ms
Query 3	720 ms	703 ms

Figure 6.7: Results of last four developers

```
// get ws courts of lisbon
External_Court[] courts = courts_webservice.GetByCourt_City("Lisboa");

// result list
List<External_Judge> judgesList = new List<External_Judge>();

foreach (External_Court court in courts)
    judgesList.AddRange(judges_webservice.GetByCourtName(court.Court_Name));
```

```
from wsJudge in judgesWS
join wsCourt in courtsWS on wsJudge.CourtName equals wsCourt.Court_Name
where wsCourt.Court_City == "Lisboa"
select wsJudge;
```

Figure 6.8: Development effort: query 1

```
    // get all ws courts
    External_Court[] wscourts = courts_webservice.GetAll();

    // get all ws judges
    External_Judge[] wsjudges = judges_webservice.GetAll();

    // result list
    List<OSUSR_D95_CLIENT11> emailsList = new List<OSUSR_D95_CLIENT11>();

    List<External_Judge> foundJudges = null;
    string inList = null;
    foreach (External_Court c in wscourts)
    {
        foundJudges = FindJudgesForCourt(wsjudges, c);

        inList = "(";
        foreach (External_Judge found in foundJudges)
            inList += "'" + found.Judge + "', ";

        inList = inList.Remove(inList.LastIndexOf(",")) + ")";
        emailsList.AddRange(new databaseDataContext().ExecuteQuery<OSUSR_D95_CLIENT11>(
            "SELECT * FROM OSUSR_D95_CLIENT11 WHERE NAME IN " + inList, new object[] { }));
    }
}

private static List<External_Judge> FindJudgesForCourt(External_Judge[] wsjudges, External_Court c)
{
    List<External_Judge> result = new List<External_Judge>();

    foreach (External_Judge j in wsjudges)
        if (j.CourtName.Equals(c.Court_Name))
            result.Add(j);

    return result;
}
```

Figure 6.9: Development effort: query 3 (developer)

```
from wsCourt in courtsWS
join wsJudge in judgesWS on wsCourt.Court_Name equals wsJudge.CourtName
join dbUser in usersTable on wsJudge.Judge equals dbUser.NAME
select dbUser.EMAIL;
```

Figure 6.10: Development effort: query 3 (optimizer query engine)

7

Conclusions

This project focused on studying the topics addressed in this document and to propose a solution for *OutSystems Agile Platform* regarding the optimization of queries over databases and web services. Hence, taking into account the model and algorithms we presented and validated, we formulate a proposal to *OutSystems*, specifying what has to be included in their model and some interface suggestions about how to represent the new features.

We achieved the goals proposed for this project, that had the main focus on allowing *Agile Platform* to execute queries over databases and web services with an efficient performance. With our solution, it is simple to adapt the platform to support the model and the algorithms we presented, as we describe next. In addition, the costs and the benefits of our solution became much more interesting than what expected in the beginning of this project.

This proposal focuses on two components of *Agile Platform*: the IDE used by *OutSystems* developers, *ServiceStudio*, and the application server *ServiceCenter*, where some optimizer metrics can be captured.

7.1 Model proposal

Some variables defined in the system should be configurable by someone with appropriate domain knowledge. These variables should be controlled because for certain application environments, adapting the optimizer contributes to a better query execution efficiency:

- *SLIDING_WINDOW*: contained inside class *Entity*, represents the maximum number of values gathered by the system for the moving averages (queries/APIs time costs and retrieved rows). The higher this value is, the more precise the averages are.

- *DEFAULT_ENTITY_SIZE*: contained inside class *Entity*, this value is used by the system to consider the size of an entity when no related statistic or hint metric is available.

- *DIFFERENCE_ALLOWED_FOR_API_CHANGE*: the delay in milliseconds associated with APIs that retrieve all the records (recall section 5.4.2), used by the optimizer when computing the best query/API to invoke for an entity.

Internally, every feature explained in this document should be implemented, such as the underlying statistical and hints model, the data structures for the graph, and the execution algorithm contained in the optimizer query engine.

Developers use *ServiceStudio* to develop IT projects with *Agile Platform*. Therefore, they should be able to supply the hints to the optimizer in the project, regarding entities and its columns. In order to do that, the classes representing the structure of the data arising from the data sources should exist, so developers can populate those hints. When a database table is created in a project, *ServiceStudio* generates an entity which becomes visible in the project. When a reference to a web service is added to the project, a memory structure is added in the project representing the structure of the web service, as you can see in Figure 7.1. Moreover, an object containing the APIs available for the web service is generated in the logic layer of the project, as shown in Figure 7.2. Internally, the class representing this last object should extend our model class *ExternalEntity*, in order to connect the web service with our statistics model.

Having database entities and the structure for web services already represented in the project, developers may now supply hints over these entities and their attributes. Besides, the naive naming convention we followed for the web service APIs should now be implemented in a coherent way. Therefore, we take advantage of the knowledge of the developer and he must explicitly specify which API fetches all the records from the web service, as well as which APIs fetch records via indexed attributes. It is possible that some of these APIs are not offered by the web service and thereby he may specify that. However, the system should not allow the use of web services that do not offer at least an API to fetch all of its records, or an API to fetch data by a specific attribute. In such cases, an error should show up in the application error console.

Primary keys have a strong relevance in *Agile Platform*. Although we do not present primary keys in our model, they can be simulated by classifying the attribute as unique and without any null value. For each web service, a field identifying which is its primary key should be specified in the structure object. Specifying a primary key attribute is

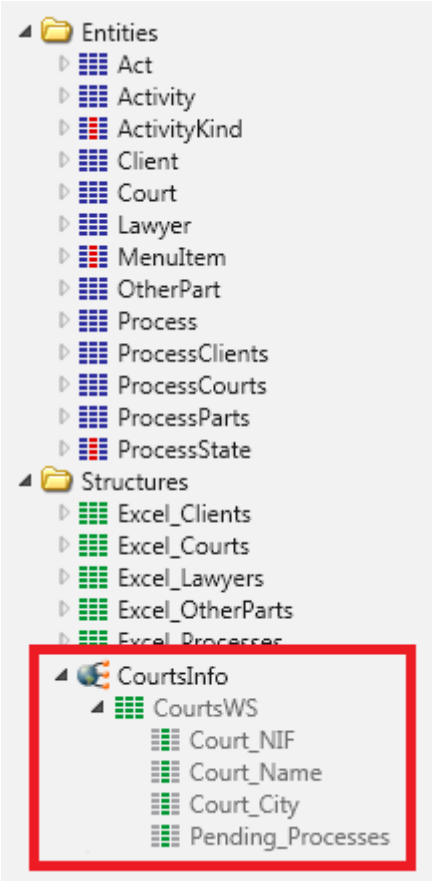


Figure 7.1: Web service structure generated in *ServiceStudio*

equivalent to consider such attribute as unique and mandatory (no nulls allowed). In our model, for an attribute of an entity, we represent the metric *mandatory* as 0% of null values in that attribute. Thus, we propose the possibility for a developer to fill a mandatory property for every entity attribute, meaning that such attribute has 0% of null values.

For every entity, either it is a database table or a web service, a developer can supply a hint regarding its dimension. Besides, for web services, he must also specify which attribute is its primary key. Finally, he must also refer to the existence of an API method fetching all the records from the web service. When such method exists and it has the correct signature, it is detected and presented in the respective dropdown box, otherwise the developer must create an extension in order to allow its invocation. Figure 7.3 shows the information that may be supplied to the web service of courts, generated in *Service Studio*.

When selecting the API to fill in *Get All*, a dropdown box appears where he can choose the related API, or create a new extension action if such API do not exist. He may also specify that the web service does not contain such API. These features are shown in Figure 7.4. These extensions may have to be created when the API offered by the web service includes more input parameters than a single web service attribute, such as the example

Figure 7.2: Web service APIs generated in *ServiceStudio*

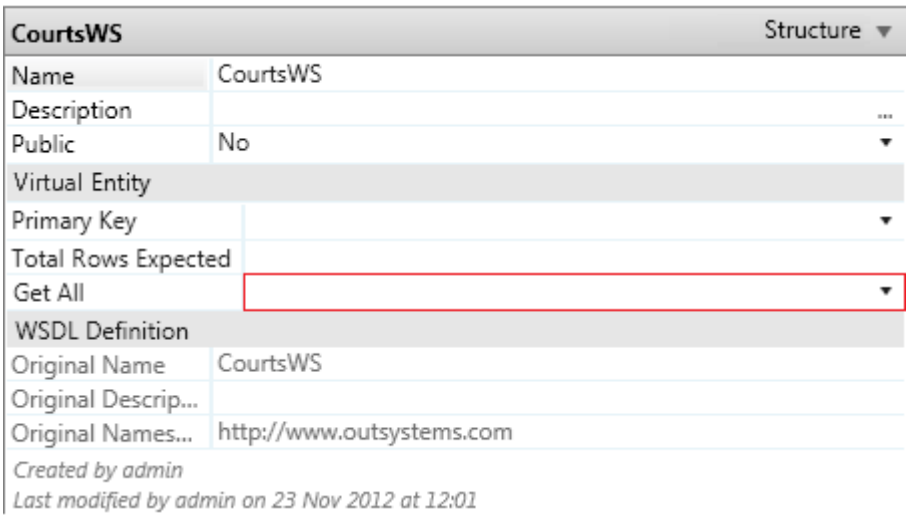


Figure 7.3: Virtual entity

of APIs with authentication. For those cases, considering that the authentication information is stored in session variables, the extension should invoke the API by supplying the necessary input parameters.

As for the hints over the attributes of entities, these should be supplied in the section *Virtual Entity Attribute*, when selecting an attribute of the entity's structure, as presented in Figure 7.5. If the developer specifies that an attribute is unique, the column for the expected percentage of distincts should be blocked. Moreover, it only becomes possible to fill the percentage of expected null values when the property *Is Mandatory* has the value *No*. When clicking in the dropdown box of *Foreign Key To*, *Service Studio* should show all the entities' identifiers (primary keys) and secondary identifiers (unique attributes), of both database tables and web service structures. This feature is necessary because in *Agile Platform* it is only possible to specify foreign keys to attributes containing a unique index on it, or to primary key attributes.

Finally, when specifying which API allows indexing that attribute, the dropdown box should show the API that contains the correct signature, as explained before. In the same way, if it does not exist, the developer should develop an extension to ensure the correct use of the API. This scenario is shown in Figure 7.6.

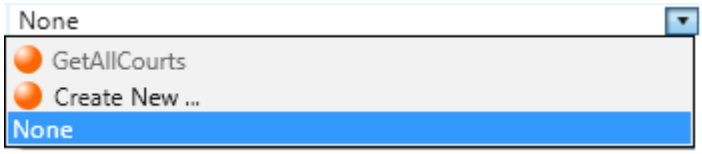


Figure 7.4: Informing the optimizer about a *GetAll* API

All these features should exist for web services and its attributes. However, for database tables, the APIs specification options do no exist, neither the primary key option, since those concepts are already captured by the nature of a database table.

The system should show a warning for every hint not populated. As presented in our model, the optimizer execute queries in the absence of statistics nor hints, although with less efficiency. Therefore, by populating every hint, developers ensure that the optimizer will always have a helpful information base.

We abstracted web services and database tables as entities. To mix these data sources in *OutSystems* queries, web service entities could be included in the existing query operator *SimpleQuery*. In order to do that, our model should be able to implement all the expressiveness features already present in the actual *SimpleQuery*. Therefore, improving our model until it handles all the query operators allowed in *SimpleQueries* is part of our future work, as we reveal in the next section. Afterwards, database entities and web services could be mixed in *SimpleQueries* and queries executed with the algorithm presented in chapter 4.

7.2 Future Work

As for the future, we aim to solve and improve some of the constraints presented in the beginning of chapter 4.2. However, we do not address constraints or problems arising from the expressiveness of the query language Linq, since the main goal of this project is to propose a solution for *OutSystems Agile Platform*. Therefore, only restrictions over our model will be addressed in the future and we summarize them below:

1. More join strategies implemented in the optimizer for merging collections in memory

2. More types of joins considered for the queries. Include left outer joins, right outer joins and full outer joins, which implies studying its differences and other optimizations techniques when facing them.

3. Improve the web services APIs names nomenclature, possibly by forcing a developer to specify the API fetching all records from the web service and the indexed APIs, as presented in the previous section.

4. Improve the complexity of web services considered, more specifically regarding their input interface, by considering several input arguments. By considering APIs

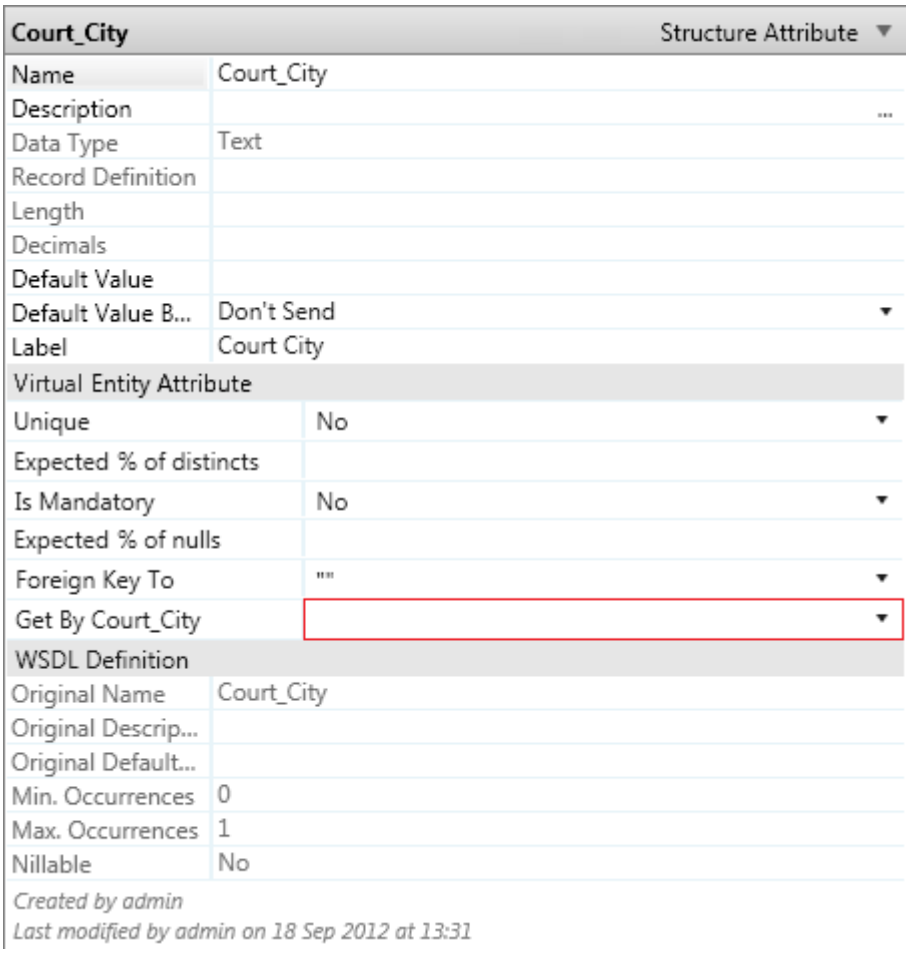


Figure 7.5: Populating information of attributes

with several input arguments, we include the concept of composite indexes for web services.

5. Include more query operators such as group by, order by, as well as more filtering operators like *OR*, > and <. These features have impact on the way of fetching data from the entities and on interesting orders maintained on the data sets to execute queries faster.

6. Load database entities statistics from the database catalog and maintain averages of times and rows for their queries, implementing efficient query recognition methods.

7. Adapt the algorithm to be able to automatically change the queue execution order, by grouping interesting sequences of operations over the same database, therefore aiming at sending queries to that database just once, resulting in a better performance.

8. Improve the statistics model by including histograms over the data of indexed columns, for a more precise cost estimation of query operations.

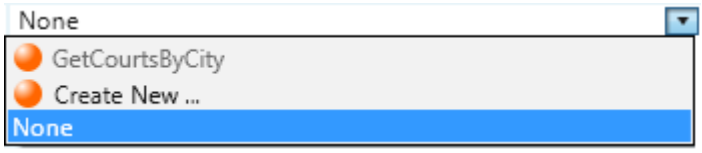


Figure 7.6: Choosing an indexed API for a web service attribute

7.3 Final remarks

Thanks to this dissertation we have learnt and mastered our knowledge regarding query optimization techniques over databases and external systems, specifically web services. In addition, we have studied alternative query execution algorithms, in order to choose a proper one that fit with our goals. Still, by developing this project inside the R&D department of *OutSystems*, we had contact with real scenarios regarding the integration of data between databases and web services, therefore greatly contributing to contextualize ourselves with real problems arising from the development of these features by teams of developers, in an enterprise. Finally, during this year of work we have improved our research skills because over the time we understood which keywords supplied us more useful and related articles, thereby allowing us to discover more suitable information that we could use to create and develop our solution.

We had some difficulties during the development of this project. At the beginning, it was hard to understood which keywords led us to the best articles and thus the preparation phase of this dissertation was not much accurate, since we did not follow much of the material studied by that time, neither the solution plan organized for the implementation phase. However, due to the successful research phase performed afterwards, and some experience gained, we successfully achieved our goals by creating an automated and efficient solution to solve the problems presented in this document. In detail, we had a hard time finding a solution to allow us to develop a query execution algorithm that let us write Linq queries. We found Re-Linq framework that allowed us to overcome this problem. Afterwards, we abstracted the common concepts of database and web service entities and we built a model where we could maintain precise statistical information that allows an optimizer query engine to precisely estimate the costs of query operations like filters and joins. Nevertheless, it took more than a month to find an appropriate and rather simple query execution algorithm that could execute our queries. When we found it, we improved and transformed it by making the algorithm adaptive, so when the entities and its data evolve and change over the time, the algorithm still computes the best execution flows and uses the most efficient queries and APIs to fetch data from the entities.

Having a framework that allowed us to implement a query execution algorithm, an organized and structured model, and a well studied query execution algorithm, the implementation phase did not reveal problems. However, when the first version of our

optimizer query engine was ready, we tested and it was still slow, due to the naive approach we were using to join the data sets in memory. Thus, we replaced such naive approach by using an indexed nested loop join strategy and the results were finally good.

We proposed a solution for *OutSystems Agile Platform* regarding the execution of queries over databases and web services. We intend to continue this work by improving the constraints presented in this document, as well as facing the topics written in section 7.2. When those constraints are no longer part of our model, our optimizer query engine will be able to execute queries closer to the expressiveness of SQL queries, therefore becoming an even greater help for developer teams that constantly need to manually write integration algorithms and be sure of their efficiency.

Bibliography

[Alf] José Júlio Alferes. Slides of the course sistemas de bases de dados, department of informatics, fct-unl.

[BH12] Richard BANISTER and Thomas Edgar HAWKES. Bi-directional replication between web services and relational databases. (US 2009/0063504 A1), February 2012.

[BQ08] Ulf Leser Bastian Quilitz. *Querying Distributed RDF Data Sources with SPARQL*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM.

[DC10] Suzanne W. Dietrich and Mahesh Chaudhari. The linq between xml and databases: a gentle introduction. *J. Comput. Sci. Coll.*, 25(4):158–164, April 2010.

[DSD95] Weimin Du, Ming-Chien Shan, and Umeshwar Dayal. Reducing multidatabase query response time by tree balancing. *SIGMOD Rec.*, 24(2):293–303, May 1995.

[EDNO97] Cem Evrendilek, Asuman Dogac, Sena Nural, and Fatma Ozcan. Multidatabase query optimization. *Distrib. Parallel Databases*, 5(1):77–114, January 1997.

[Ell02] Kline Rodger N. Ellis, Nigel R. Automatic database statistics creation. (US 2002/0087518 A1), July 2002.

[Fab] Fabian. Linq: A good visitor use case (but bad implementation). https://www.re-motion.org/blogs/mix/2010/04/18/linq-a-good-visitor-use-case-but-bad-implementation/.

[Gie] Markus Giegl. re-linq|ishing the pain: Using re-linq to implement a powerful linq provider on the example of nhibernate. http://www.codeproject.com/Articles/42059/re-linq-ishing-the-Pain-Using-re-linq-to-Implement.

[GS07] R. Guravannavar and S. Sudarshan. Reducing order enforcement cost in complex query plans. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 856 –865, april 2007.

[GSS07] P.B. Guttoski, M.S. Sunye, and F. Silva. Kruskal's algorithm for query tree optimization. In *Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International*, pages 296 –302, sept. 2007.

[HM04] Elliotte Rusty Harold and W. Scott Means. *Xml in a nutshell, 3rd edition*. O'Reilly Media, Inc., 3 edition, 2004.

[Kos00] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, December 2000.

[Mama] Margarida Mamede. Slides of the course algoritmos e estruturas de dados, department of informatics, fct-unl.

[Mamb] Margarida Mamede. Slides of the course análise e desenho de algoritmos. http://orium.homelinux.org/univ/lei/ada/.

[Mic] Microsoft. Linq query samples. http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b.

[RGL90] Arnon Rosenthal and Cesar Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. *SIGMOD Rec.*, 19(2):291–299, May 1990.

[RHpt] N.A. Rakhmawati and M. Hausenblas. On the impact of data distribution in federated sparql queries. In *Semantic Computing (ICSC), 2012 IEEE Sixth International Conference on*, pages 255–260, Sept.

[SBB03] D.E. Shasha, P. Bonnet, and P. Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 2003.

[Sch] Fabian Schmied. re-linq: A general purpose linq foundation. https://www.re-motion.org/download/re-linq.pdf.

[sita] American national standards institute standardized structured query language. http://www.ansi.org/.

[sitb] Creating temporary tables dynamically with linq. http://stackoverflow.com/questions/337704/parameterizing-an-sql-in-clause/337864#337864.

[sitc] Limited efficiency of sql in (list) queries. http://www.techrepublic.com/article/build-temporary-tables-to-optimize-sql-queries/5796615.

[sitd] World wide web consortium. http://en.wikipedia.org/wiki/W3C.

[SKS10] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2010.

[SMWM06] Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Query optimization over web services. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 355–366. VLDB Endowment, 2006.

[SO95] D.D. Straube and M.T. Ozsu. Query optimization and execution plan generation in object-oriented data management systems. *Knowledge and Data Engineering, IEEE Transactions on*, 7(2):210 –227, apr 1995.

[Wal07] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

Appendix

A.1 Creating a web service connection with Linq

We present the set of steps needed to create a web service connection with Linq. Figure A.2 shows how the project containing the provider should look like.

1. Add a web service reference to the project under "Service References", Figure A.2
2. Implement the interfaces *IQueryable<T>*, *IOrderedQueryable<T>*, and *IQueryProvider*, needed for any Linq provider (classes *QueryableCourtsServerData.cs* and *CourtsServerQueryProvider.cs*, Figure A.2)
3. Add a custom .NET type to represent the data arising the web service, Figure A.1
4. Create a query context class that executes an expression tree that is passed to it (class *CourtsServerQueryContext.cs*, Figure A.2)
5. Create a class that obtains the data from the web service. It contains the calls to the web service APIs (class *WebServiceHelper.cs*, Figure A.2)
6. Create an expression tree visitor subclass that finds the expression that represents the innermost call to the Queryable.Where method (class *InnerMostWhereFinder.cs*, Figure A.2)
7. Create an expression tree visitor subclass that extracts information from the Linq query to use in the Web service request (class *CourtsFinder.cs*, Figure A.2)
8. Create an expression tree visitor subclass that modifies the expression tree that represents the complete Linq query (class *ExpressionTreeModifier.cs*, Figure A.2)

```
public class WS_Court
{
    // Properties.
    public string City { get; private set; }
    public string NIF { get; private set; }

    // Constructor.
    internal WS_Court(string city,
                      string nif)
    {
        City = city;
        NIF = nif;
    }
}
```

Figure A.1: Adding a custom .NET type

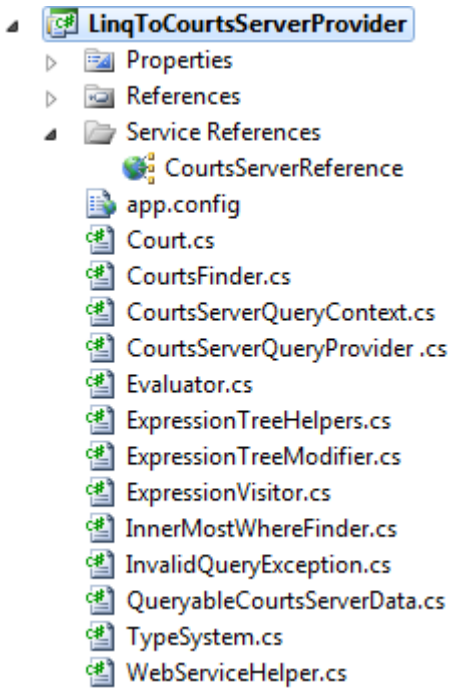


Figure A.2: Web service provider component

9. Use an evaluator class to partially evaluate an expression tree. It translates all local variable references in the Linq query into values (class *Evaluator.cs*, Figure A.2)

10. Create an expression tree helper class and a new exception class (classes *Expression-TreeHelper.cs* and *InvalidQueryException.cs*, Figure A.2)

A.2 Building on Re-Linq

A.2.1 Building Re-Linq sources

We started by learning how to use Re-Linq and we created a project where we could use Re-Linq to start implementing a query execution algorithm. Re-Linq is an open-source project and its sources are available to be downloaded from re-linq codeplex project page[1]. To our project we used the version *1.13.164*. Once the binary sources were downloaded, we created a class library project and included a reference to the following binaries:

- *Remotion.Linq.dll*
- *Remotion.Linq.SqlBackend.dll*
- *Remotion.Linq.LinqToSqlAdapter.dll*

Afterwards, in order to begin the creation of our query provider, we implemented the necessary interfaces described in section 3.6.2.1, on the following classes:

- *ProviderQueryable.cs*
- *ProviderQueryExecutor.cs*

Then, we added our data sources to the project. Database entities are added like described in section 3.10 and web services like described in section 3.6.1.

Finally, in order to start developing the algorithm, we implemented the method *ExecuteCollection<T>* in *ProviderQueryExecutor.cs* class. To test the tool, we created a separate console application within the general solution and we added the binary *Remotion.Linq* and the library project to its references. After completed these steps, we were able to write queries to be executed by our tool. Figure A.3 shows the class with the queries being executed by our tool, where the object *ProviderQueryFactory* is simply an entry point for the creation of our provider.

A.2.2 Creating a Linq provider with Re-Linq

We present in Figures A.4 and A.5 the classes *ProviderQueryable* and *ProviderQueryableExecutor*, necessary to the creation of our query provider.

- *ExecuteCollection<T>* is called for queries returning a collection of items. It receives a *QueryModel* as input argument.

[1] http://relinq.codeplex.com/

```
namespace OptimizedExecutor.Tests
{
    class Program
    {
        static void Main(string[] args)
        {

            /********************** DATA SOURCES *************************/

            // DATABASES
            var courtsTable = ProviderQueryFactory.Queryable<OSUSR_D95_COURT11>();
            var usersTable = ProviderQueryFactory.Queryable<OSUSR_D95_CLIENT11>();

            // WEB SERVICES
            var courtsWS = ProviderQueryFactory.Queryable<Courts_WS.CourtsWS>();
            var citiesWS = ProviderQueryFactory.Queryable<Cities_WS.Cities_WS>();
            var judgesWS = ProviderQueryFactory.Queryable<Judges_WS.Judges_WS>();

            /************************ QUERIES ***************************/

            var mergeQuery = from dbCourt in courtsTable
                             join wsCourt in courtsWS on dbCourt.NAME equals wsCourt.Court_Name
                             select wsCourt;


            /************************** RUNS ********************************/
            foreach (var c in mergeQuery)
                Console.WriteLine(c);
```

Figure A.3: Testing the tool

- *ExecuteSingle<T>* is called for queries returning a single item from a collection. It receives a *QueryModel* as input argument with a *ResultOperator* attached to it (eg. *First()*, *Last()*, *Min()*). Even when these operators return a scalar value because the query returns a sequence of scalar values, they still invoke *ExecuteSingle<T>* because a single item is chosen from the list, rather than calculated.

- *ExecuteScalar<T>* is called for queries returning a scalar value, calculated from the result sequence of the query. It receives a *QueryModel* as input argument with a *ResultOperator* attached to it (eg. *Count()*, *Sum()*)

A.2.3 Context about *QueryModels*

We present with more detail the structure of a Re-Linq *QueryModel*. Figure A.7 is taken from the debugging session of VisualStudio 2010, for the Linq query A.6.

From the attributes inside the *QueryModel*, the relevant ones are the clauses in the query. *MainFromClause* holds a representation of the first clause in the query: *"from wsCourt in courtsWS"*. It contains metadata information about the type of the data source being considered (*itemType*), a Re-Linq expression representing the access to the data source

```
/// <summary>
/// Provides the main entry point to a LINQ query.
/// </summary>
public class ProviderQueryable<T> : QueryableBase<T>
{
    private static IQueryExecutor CreateExecutor()
    {
        return new ProviderQueryExecutor();
    }

    // This constructor is called by our users, create a new IQueryExecutor.
    public ProviderQueryable()
        : base(QueryParser.CreateDefault(), CreateExecutor())
    {
    }

    // This constructor is called indirectly by LINQ's query methods, just pass to base.
    public ProviderQueryable(IQueryProvider provider, Expression expression)
        : base(provider, expression)
    {
    }
}
```

Figure A.4: Implementing *QueryableBase<T>*

(*fromExpression*) and the name of the item (*itemName*), as you can see in Figure A.8. *BodyClauses* is an array containing 1 or more *BodyClause*. Those can be of 2 kinds: join clauses (*JoinClause*) or where clauses (*WhereClause*). Finally, a *SelectClause* holds the output structure information related to what was specified in the select clause of the query.

From the several kinds of existing Re-Linq expressions, we deal with the following:

- *QuerySourceReferenceExpression*: an expression representing a reference to an entity, Figure A.9.
- *MemberExpression*: an expression representing an access to a property of an entity, Figure A.10.
- *BinaryExpression*: an expression representing a comparison between 2 or more expressions, Figure A.11.
- *ConstantExpression*: an expression representing a value, Figure A.12.
- *NewExpression*: an expression representing an anonymous type that has one or several expressions inside, Figure A.13.

We do not deal with sub-queries, neither with user-defined or system functions and therefore dealing with *SubQueryExpression* and *MethodCallExpression* in our tool is not addressed.

```
// Called by re-linq when a query is to be executed.
public class ProviderQueryExecutor : IQueryExecutor
{

    // Executes a query with a scalar result, i.e. a query that
    // ends with a result operator such as Count, Sum, or Average.
    public T ExecuteScalar<T>(QueryModel queryModel)
    {
        return ExecuteCollection<T>(queryModel).Single();
    }

    // Executes a query with a single result object, i.e. a query that
    // ends with a result operator such as First, Last, Single, Min, or Max.
    public T ExecuteSingle<T>(QueryModel queryModel, bool returnDefaultWhenEmpty)
    {
        return returnDefaultWhenEmpty ? ExecuteCollection<T>(queryModel).SingleOrDefault()
               : ExecuteCollection<T>(queryModel).Single();
    }

    // Executes a query with a collection result.
    public IEnumerable<T> ExecuteCollection<T>(QueryModel queryModel)
    {
        throw new NotImplementedException();
    }

}
```

Figure A.5: Implementing *IQueryExecutor*

```
from wsCourt in courtsWS
join wsJudge in judgesWS on wsCourt.Court_Name equals wsJudge.CourtName
where wsCourt.Court_City == "Lisboa"
select wsCourt;
```

Figure A.6: Courts of Lisbon with judges

A.2.4 Executing a web service API

Figure A.14 shows a code snippet regarding the invocation of the API *GetByCourt_City* of the web service *WS_Courts*.

A.3 Optimizer query engine implementation

A.3.1 Parsing a *QueryModel*

We present Figure A.15 that shows simplified code of *GraphGenerator*. Figure A.16 shows the several sub-phases after generating the main structure for the graph. They create new possible connections between elements and populate the statistical metrics over the

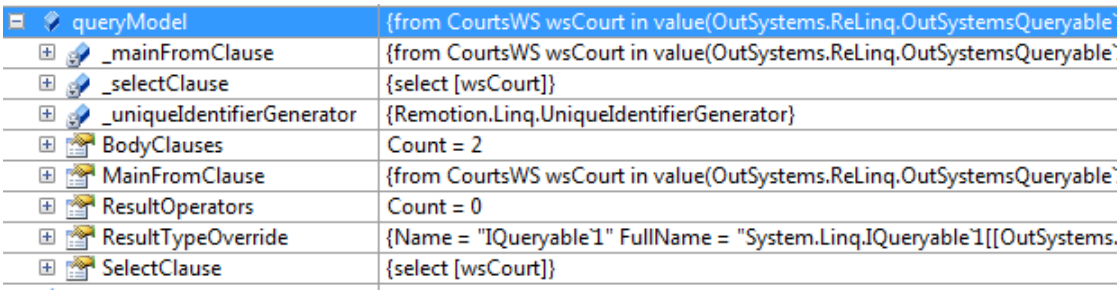


Figure A.7: Inside of a QueryModel

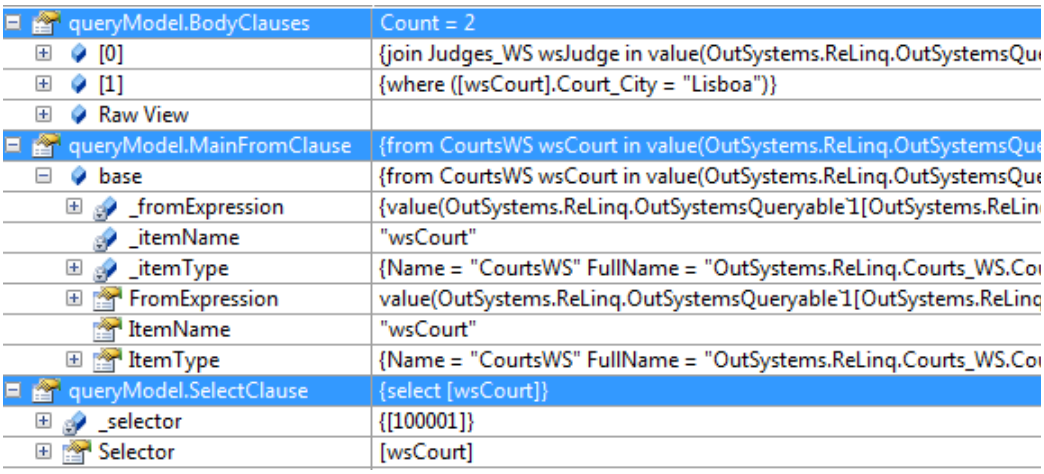


Figure A.8: Inside of a QueryModel

elements of the graph.

A.3.1.1 Exploring commutativity

The aim of exploring the commutativity between operators is to find new possible connections in the graph and therefore create and represent them. By doing this, we may discover alternate ways to execute a query, rather than the one established by the written query, that may be more efficient.

We explore commutativity between joins. Hence, we follow the basic commutative association, true for inner joins, to discover relations between joins:

$$A = B \ \& \ B = C \ \Rightarrow \ A = C$$

The query presented in Figure A.17 shows a possible scenario. The joins we have in the query are:

- *wsCourt.Court_Name = dbCourt.NAME*
- *wsCourt.Court_Name = wsJudge.CourtName*

```
select wsCourt
```

Figure A.9: *QuerySourceReferenceExpression*

```
select wsCourt.Court_Name
```

Figure A.10: *MemberExpression*

Thereby, the new join connection discovered is: *dbCourt.NAME = wsJudge.CourtName*, and the resulting graph looks like Figure A.18.

The function *exploreCommutativities* shown in Figure A.16 implements these features.

A.3.1.2 Percolating filters

Now that all join connections are created, we do the same for filters. We needed to explore the commutativity between joins in first place because for filters we try to discover any possible relation with other entities, through the available joins on the graph. By this, we mean that if a filter is being applied on an entity and it can also be applied on other entities, we represent those possibilities. Like explained before, the tougher operations on these queries are the joins. Thereby, we want to reduce the amount of records that are passed to join operations and, by trying to explore these commutativities, we try to apply the maximum number of possible filters on the data sets retrieved by the data sources, thus reducing the cardinality of the data sets before joins operations.

If there is a filter over an attribute of an entity and a join condition using such attribute, it means that we can also apply the filter to the other entity present in the join condition. Moreover, for new associations discovered, the same process is repeated. To better understand this, consider the following example:

- Filter: *wsCourt.Court_Name == "LisboaTT"*
- Join condition: *wsCourt.Court_Name == dbCourt.NAME*
- Join condition: *dbCourt.NAME == wsJudge.CourtName*

Following the simple associativity rule presented before, we have the possible filters to apply:

- *wsCourt.Court_Name == "LisboaTT"*
- *dbCourt.NAME == "LisboaTT"*
- *wsJudge.CourtName == "LisboaTT"*

```
where dbCourt.NAME == "Barreiro"
```

Figure A.11: *BinaryExpression*

```
where dbCourt.NAME == "Barreiro"
```

Figure A.12: *ConstantExpression*: "Barreiro" - right part

Consider query A.19 and the resulting graph A.20, for a concrete scenario of the application of filters percolation.

The function *percolateFilters* shown in Figure A.16 implements these features.

A.3.1.3 Generating queries and APIs

In this phase, *GraphGenerator* iterates all the filters in the graph and, for each, it generates all the suitable queries/APIs that may be invoked for that entity. Moreover, it populates the expected time cost, the expected rows, and the selectivity associated to the APIs. The selectivity metric we use for an API is the column selectivity (recall section 3.5), computed for the column associated with the indexed attribute via the API. For instance, for the *GetByCourt_City* API of *WS_Courts*, we compute the column selectivity of the column *Court_City*. For the API *GetAll()*, the selectivity is 0, since it returns all records.

These metrics are loaded from the statistic collections existing within the entities classes and thereby *GraphGenerator* consults the respective inspectors to extract such information. For queries, these costs are not populated, as explained before. Hence, for web services, an API object contains the name of the API, the input parameter, the expected time cost, the expected rows and the selectivity, while for database entities it only contains the kind of query it represents.

Consider the filters:

- *dbCourt.CITY = "Lisboa"*
- *wsCourt.Court_Name = "LisboaComercio"*

The first filter is being applied over a database entity and thereby it represents a query like *SELECT * FROM DBCourt WHERE CITY = 'Lisboa'*. On the other hand, the second filter is applied over a web service and it may represent an API, if it exists. If an API *GetByCourt_Name* is available in the web service API class (*WS_Courts* in this case), *GraphGenerator* creates a new API object and adds it to the collection *apis*, otherwise it does not generate any API object.

After all the filters of an entity have been checked, *GraphGenerator* adds queries/APIs retrieving all the elements from that entity, to its APIs collection. For database entities, we represent them with the name *FullScan*, while for web services *GraphGenerator* searches for an available *GetAll* method and adds it, if available.

```
select new
{
    dbCourt.NAME,
    dbUser.EMAIL,
    dbUser.ADDRESS,
    wsCourt.Court_NIF
};
```

Figure A.13: *NewExpression*

```
private External_Court[] GetByCourt_City(string city)
{
    Courts_WS.CourtsWS[] externalResults = null;

    Stopwatch sw = new Stopwatch();
    sw.Start();
    externalResults = client.GetCourtsByCity(city);
    sw.Stop();
    double elapsedTime = sw.Elapsed.TotalMilliseconds;

    // update api statistics
    this.AddUpdateCallAvgTime("GetByCourt_City", elapsedTime);
    this.AddUpdateCallRows("GetByCourt_City", externalResults.Length);

    External_Court[] courts = new External_Court[externalResults.Length];
    for (int i = 0; i < externalResults.Length; i++)
        courts[i] = new External_Court(externalResults[i].Court_City,
            externalResults[i].Court_Name, externalResults[i].Court_NIF);

    return courts;
}
```

Figure A.14: Invoking an API of *WS_Courts.cs*

Finally, the last kind of API objects are created. These take into account possible available data (from other entities) and use them, if suitable, to invoke APIs with certain input arguments coming from available data.

To better understand all this process, consider the join between two web services *WSCourts* and *WSJudges* shown in Figure A.21 and the generated APIs for the entities in Figure A.22. Due to the lack of space to represent the queries/APIs for every entity, we decided to represent them in a different image. Entities are represented in the same way, while APIs are represented with the notation of filters, and they have statistical information populated in their arcs. In order to invoke the API *GetByCourt_Name(wsJudge.CourtName)* of *wsCourt*, it is implicit that data from the entity *wsJudge* is available, which may not be true. These verifications are performed by the optimizer during query execution, when selecting the best query/API to invoke for the entity, taking into account possible available data. As for the statistical information annotated, there are three measures:

```
for (int i = 0; i < originalModel.BodyClauses.Count; i++)
{
    source = originalModel.BodyClauses[i];
    joinClause = source as JoinClause;
    whereClause = source as WhereClause;

    if (joinClause != null)
    {
        newVertex = buildEntityVertex(...);
        graph.addVertex(newVertex);
        found = graph.findVertex(...);

        if(found == null)
            throw new ParsingException("Error while parsing query to graph.");

        jArc = new JoinArc(found, newVertex, Expression.Equal(...));
        graph.addJoinArc(jArc);
    }
    else if (whereClause != null)
        placeFiltersInGraph(graph, buildFilters(graph, whereClause));
}
```

Figure A.15: Generating a query graph

```
exploreCommutativities(graph);
percolateFilters(graph);
AddEntitiesAPIs(graph);
PopulateArcsCosts(graph);
UpdateOutputStructures(graph);
graph.OutputStructure = ToolHelper.GenerateOutputStructure(originalModel.SelectClause);
```

Figure A.16: Generating a query graph

- the selectivity associated to the column indexed by the query/API
- the average time cost maintained for the query/API
- the average returned rows maintained for the query/API

Considering that the execution uses *GetByCourt_City("Lisboa")* to get the data from *WSCourts* in first place, when deciding how to fetch the data from *WSJudges*, it has two possibilities: a *GetAll()* API or a *GetByCourt(string court_name)* API, supplying the right argument from the available data set of courts. Hence, all the possibilities for fetching data from the entities are generated, so the optimizer can later decide which option it follows to fetch the data. The function *addEntitiesAPIs* shown in Figure A.16 is the one responsible for implementing these features.

A.3.1.4 Populating arcs costs

In this phase, *GraphGenerator* populates costs in every arc of the graph. Different metrics may be applied to this cost. We chose to use the expected number of records resultant

```
from wsCourt in courtsWS
join dbCourt in courtsTable on wsCourt.Court_Name equals dbCourt.NAME
join wsJudge in judgesWS on wsCourt.Court_Name equals wsJudge.CourtName
select dbCourt;
```

Figure A.17: Exploring join commutativity

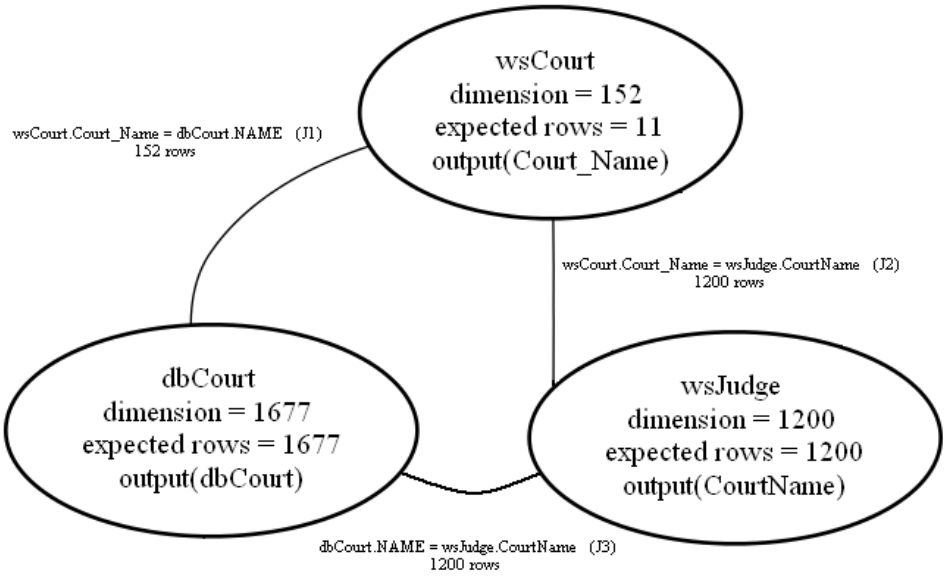


Figure A.18: Resulting graph

from operations as the cost metric for the arcs because we aim at reducing the sizes of resulting collections at every execution step and therefore the optimizer will follow the lowest costs in the graph. In spite of not knowing what is being computed on the web services side, usually, if an API retrieves less records, it means it is faster [Ell02].

To begin, *GraphGenerator* checks all the filters and populates their costs. Populating the expected number of rows of a filter is different from the expected number of rows of an API. While for an API *GraphGenerator* only consults the moving averages maintained, for filters it consults the moving averages and all the statistics and hints maintained over the columns of the entity, following the hierarchy of decisions we describe now. If one of the next measures can be computed, the computation is done, the cost is updated and the hierarchy finishes, otherwise it continues until a specified measure is computed.

1. If the indexed column is unique, the cost is 1.
2. If there is an API related with the filter and if there is an average of rows maintained over that API, the cost is the value of that average.
3. If there are distinct ratios maintained over the column (either as a statistic or a hint), the cost is computed, according to the material presented in section 3.5.
4. If there are null ratios maintained over the column (either as a statistic or a hint), the cost is computed, according to the material presented in section 3.5.

```
from wsCourt in courtsWS
join dbCourt in courtsTable on wsCourt.Court_Name equals dbCourt.NAME
join wsJudge in judgesWS on dbCourt.NAME equals wsJudge.CourtName
where wsCourt.Court_Name == "LisboaTT"
select dbCourt;
```

Figure A.19: Percolating filters

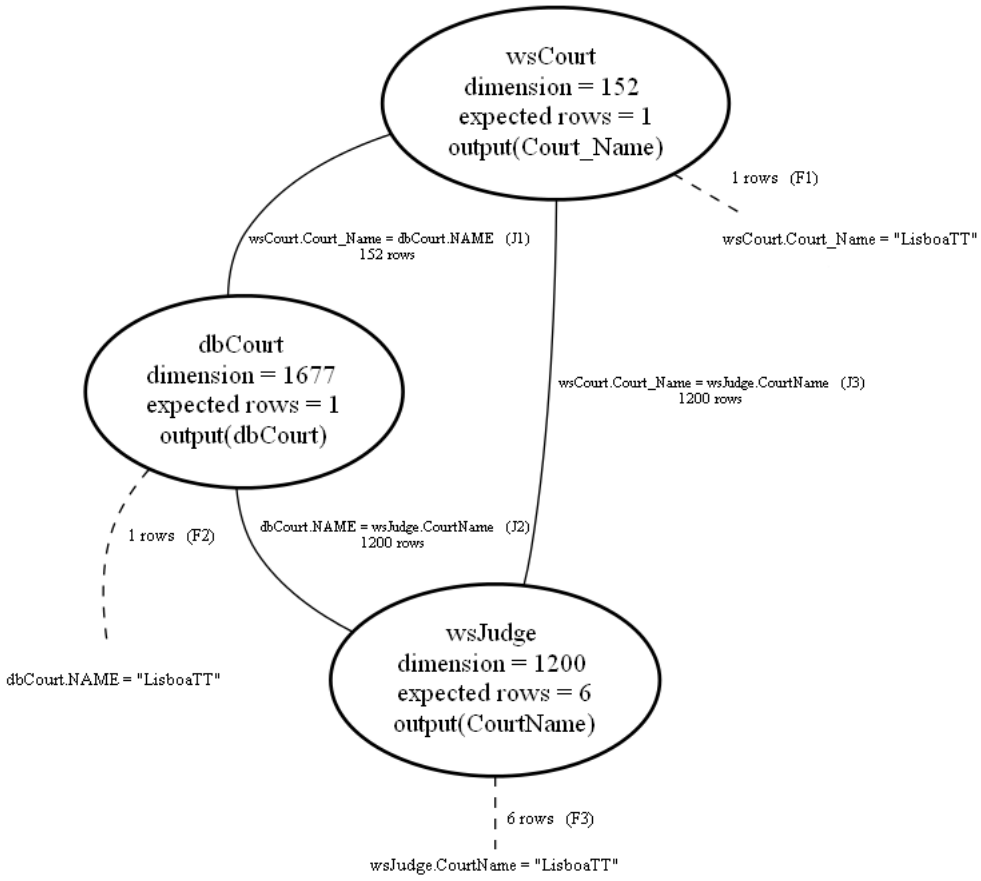


Figure A.20: Percolating filters, resulting graph

5. No measures available, the cost is the total number of rows in the entity.

Every time a statistic is consulted, for instance the number of distinct values on a column, if it does not exist, the system checks for an available hint regarding that metric and returns it, if available. When a specific metric is not available, the system returns *-1* and the hierarchy continues. The worst case when populating a cost of a filter is when no metrics are available and thereby the cost stays the total number of rows of the entity. This means that the system could not guess a reduced amount of tuples resultant after that filter is applied to a data set arising from that entity.

After the costs for all the filters being applied to an entity are computed, *GraphGenerator* populates the value for the metric *expectedRows* existent in that node, according to

```
from wsCourt in courtsWS
join wsJudge in judgesWS on wsCourt.Court_Name equals wsJudge.CourtName
where wsCourt.Court_City == "Lisboa"
select wsCourt;
```

Figure A.21: Query example

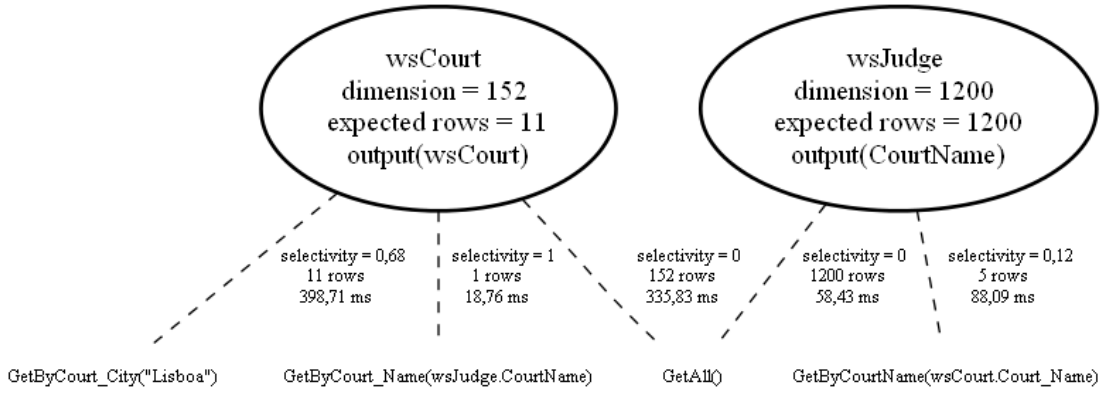


Figure A.22: Generating APIs

what presented in section 4.2.4. If there is only a single filter being applied over an entity, the value of *expectedRows* is the estimation computed for that filter. This metric is maintained over each node and it is used by our optimizer, as we explain ahead in the document.

The total number of rows is always available for database entities and web services. For database entities this metric is loaded from a database context metadata object we hold during this process, while for web services the system checks the number of rows retrieved from the *GetAll()* API, if available, or from a hint. When none of these metrics exist, the system gives a pre-defined constant to the dimension of the web service entity, which is later replaced when a statistic or a hint becomes available. This metric is represented in every node as *dimension*.

Once *GraphGenerator* populates the costs for every filter in the graph, it starts populating the costs for join arcs. In order to compute their costs, we also follow the material presented in section 3.5, taking into account the available statistics and hints maintained by the system over the columns.

The function *PopulateArcsCosts* shown in Figure A.16 is the one responsible for implementing these features.

A.3.1.5 Creating output structures

The last phase of the population of a query graph regards the output structures of each node. An output structure is the minimum set of attributes that need to be maintained from a data set to the remaining execution. When retrieving data from databases, this is

```
from wsCourt in courtsWS
join wsJudge in judgesWS on wsCourt.Court_Name equals wsJudge.CourtName
where wsCourt.Court_City == "Lisboa"
select wsCourt;
```

Figure A.23: Query example

easily controlled via an SQL query because we can specify which attributes we want to have in the result. However, when retrieving data from the web services we consider, full records are retrieved and not every attribute needs to be maintained in memory, since it wastes space.

Hence, in this phase, *GraphGenerator* iterates every node in the graph and, for each one, it consults its join arcs. The join arcs of a node are available in the data structure, as explained in section 4.2.4. Thus, *GraphGenerator* finds all the distinct attributes of the entity appearing on the join conditions of the node and adds them to the output structure. Finally, once all join arcs are verified, *GraphGenerator* consults the select clause written for the query and adds to the output structure every entity attribute specified that was not added in the previous phase.

As described in the appendix section A.2.3, we deal with certain Linq expressions. A developer is able to specify the following expressions in a *SelectClause*:

- *NewExpression*: selecting one or more attributes, or entities
- *MemberExpression*: selecting a single attribute of an entity
- *QuerySourceReferenceExpression*: selecting an entity (full records are retrieved, therefore containing all the attributes of the entity)

For better understanding this feature, consider the graph in Figure A.24, which represents the query of Figure A.23. As you can see, for the entity *wsJudge*, the only attribute which needs to be maintained is *CourtName*, since it is used on the join condition. On the other hand, all the attributes of *wsCourt* need to be held because the query *SelectClause* specifies such.

The function *UpdateOutputStructures* shown in Figure A.16 is the one responsible for implementing these features. The output structure for the graph is also created, although it is not visible in the representation. The output structure for the graph consists on what is specified in the *SelectClause* of the query and it is used to retrieved the final result when the execution ends.

A.3.2 Execution algorithm

Figure A.25 shows the main recursive function of the optimizer query engine.

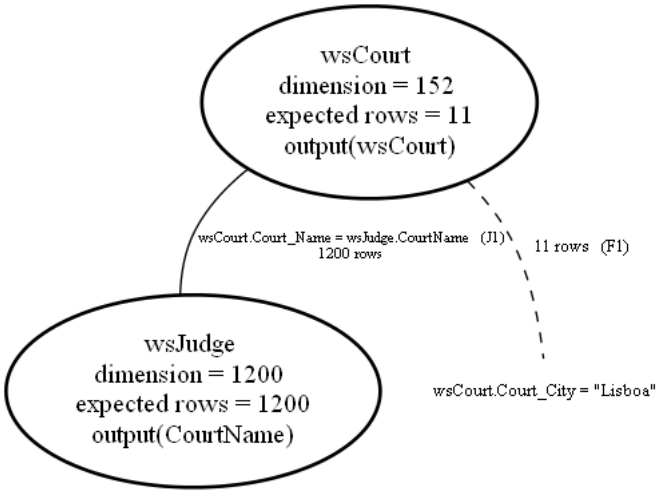


Figure A.24: Query graph representation

A.3.3 Model implementation

A.3.3.1 Statistics implementation and maintenance

We present Figure A.26 that details the class *Entity*. The collection *foreignKeys* contains foreign keys information regarding indexed columns, which may be supplied as developer hints, or loaded information from a database catalog. Both the time costs and the rows statistics are stored in dictionaries that are accessed via a string key: the API call text, such as *GetAll*, *GetByName* and so on. For a specified key, the related value is a moving average of time costs/rows retrieved by the calls.

For database entities, storing the query text as the dictionaries key is a very simple and inaccurate approach. As an example, the next two queries belong to the same type but would not be detected as such.

- *SELECT * FROM DBCourts WHERE DBCourts.NAME = "Lisboa"*
- *SELECT * FROM DBCourts WHERE DBCourts.NAME = "Porto"*

Hence, as part of future work, not only we intend to load statistics from the database catalog, but also maintain the averages of times and rows for database entities, implementing efficient query recognition methods. This may be achieved by storing query types or patterns, instead of their full SQL texts.

We also store an integer counter (*resetCounter)*, which works as a reset counter for the statistic collections. This feature is useful for environments in continuous changes, since when data changes the averages and the summaries presented also change and therefore the statistics may become inconsistent for a while. Due to this fact, a reset may be applied and the collections cleared. Furthermore, *type* is simply metadata information.

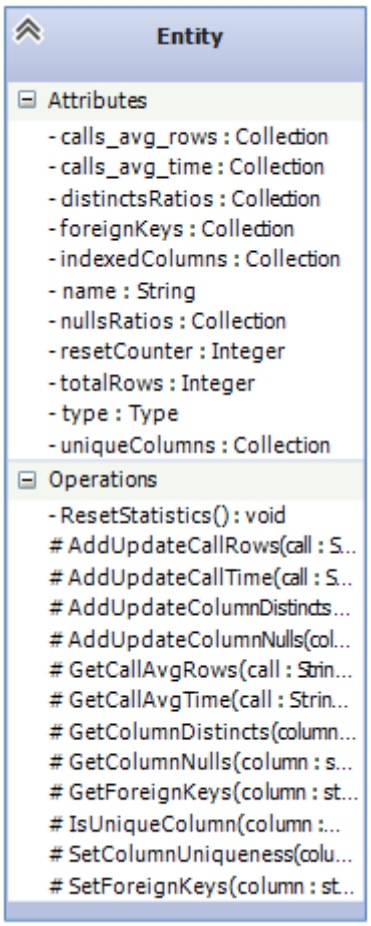
```
private void recursiveExecution(Arc toExecute)
{
    queue.Remove(toExecute);
    int tuplesRetrieved = 0;

    // arc is a filter
    if (toExecute is FilterArc)
    {
        // remove other filters applied to the same vertex
        removeQueueRelatedFilters(toExecute as FilterArc);

        // execute filter
        if ((toExecute as FilterArc).From is WSVertex)
            tuplesRetrieved = ExecuteWSFilterArc(toExecute as FilterArc).Count;
        else
            tuplesRetrieved = ExecuteDBVertex((toExecute as FilterArc).From as DBVertex).Count;

        // build index on the entity's data
        if (queue.Count > 0)
        {
            List<Vertex> filterEntityList = GetEntityList((toExecute as FilterArc).From);
            buildIndexOn(filterEntityList, GetEntityData((toExecute as FilterArc).From), ...);
        }
    }
    else if (toExecute is JoinArc) // arc is a join
    {
        removedJoinArcs.Add(ToolHelper.NormalizeExpression(toExecute.Condition, false), toExecute);
        tuplesRetrieved = ExecuteJoinArc(toExecute as JoinArc);

        // remove other filters applied to join vertexes
        removeQueueRelatedFilters(toExecute as JoinArc);
    }

    // update queue costs with the real no. tuples of retrieved
    updateQueueCosts(toExecute, tuplesRetrieved);

    if (queue.Count > 0)
    {
        // get the next arc
        Arc next = queue.First().Key;
        recursiveExecution(next);
    }
}
```

Figure A.25: Optimizer recursive algorithm

When a web service is initialized, the first step to do is to inspect its class containing the API methods via .NET reflection mechanisms, in order to populate two collections: the set of columns holding statistics (*indexedColumns*) and the collection holding the columns uniqueness measures (*uniqueColumns*).

During the investigation of an API class, the system detects the columns that can be indexed via an API call. For example, the existence of a *GetByCourt* API means that the column *Court* is indexable and therefore an entry with <*Court*, *true*> is added to the *indexedColumns* collection. Moreover, if a specific API call (*GetByCourt*, for instance) has a function output cardinality of 1, then that entity's indexed column (*Court*) is unique and therefore an entry with <*Court*, *true*> is added to the *uniqueColumns* collection, otherwise an entry with <*Court*, *false*> is added.

Afterwards, if there are statistics available in the XML documents, they should be



- *name*: name of the entity
- *totalRows*: total rows of the entity
- *indexedColumns*: collection holding the columns that have statistical information
- Statistic summaries:
  - *calls_avg_time*: collection with entity queries/calls expected times
  - *calls_avg_rows*: collection with entity queries/calls expected rows
  - *uniqueColumns*: collection containing the uniqueness measure for the entity columns
  - *distinctsRatios*: collection with the percentage of distinct values for specific columns
  - *nullsRatios*: collection with the percentage of null values for specific columns

Figure A.26: Entity class model

loaded into the appropriate collections. At this step, since the system already knows which columns should hold the statistical measures (collection *indexedColumns*), only these columns information are loaded from the available statistical XML documents. The collections created and populated at this step are:

- *distinctsRatios*
- *nullsRatios*

If a column is considered to be unique after investigating an API definition and a previous statistical file has classified it as not unique, this last measure should be ignored since the API may have changed and this statistic measure is not yet up to date.

Regarding the averages maintained over the expected number of rows and the time cost of all available APIs, these are implemented as moving averages. A moving average is a set of values tracked by the system, where the size of the set is controlled by a size. Once that size is reached, old values are discarded from the sets and new values are added to the set, in a circular way. Figure A.27 shows an example of three moving averages maintained over the times of the APIs of the web service *WSJudges*.

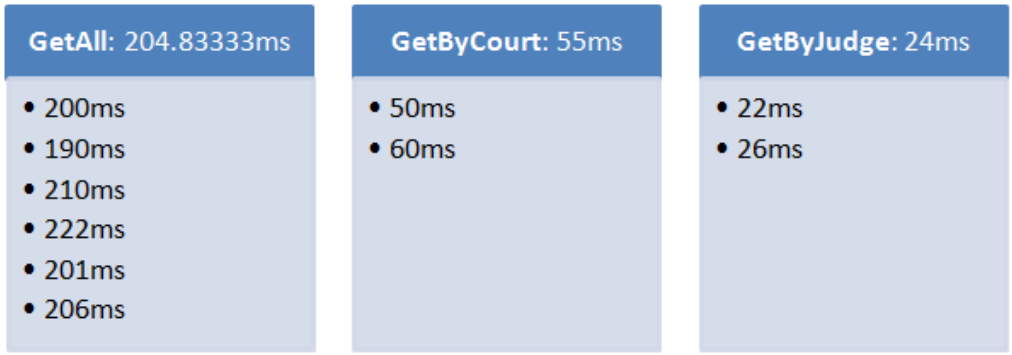


Figure A.27: Moving average example

The corresponding collections maintaining these metrics are:

- *calls_avg_rows*
- *calls_avg_times*

Maintaining some of these statistics is straight-forward. More specifically, maintaining the average rows returned by a call, as well as the average time taken, just requires accessing the collection via a string key and update the existing value, which is an arithmetic calculation. Thereby, every time an API call is invoked, the system tracks the time it takes and the number of records retrieved and updates the collections *calls_avg_time* and *calls_avg_rows*, respectively, by accessing the dictionaries via the key and replacing its value.

As for the unique columns summaries, an update on a column is done by replacing the value for a specific key, while a full collection update is done by clearing the *uniqueColumns* dictionary and re-inspecting the APIs class, populating the dictionary again.

However, the situation is different for the ratios of distincts and nulls. The system just updates these summaries when all the records from a web service are fetched (for example through a *GetAll* call), because it is the moment when it is sure of the exact distribution of columns values in the entity. By invoking a *GetBySomething* API, the system only has access to a part of the records from the entity and therefore it cannot precisely update the distribution summaries for columns. Thus, the ratio of null values in indexed columns is measured when all the records are fetched from that entity, by iterating the result collection and counting the number of null values in the columns holding the distribution statistics. These counters are stored in an auxiliary collection during the iteration and, when the iteration finishes, they are stored in the entity class collection *nullRatios*, by updating the dictionary in the related key (column name), replacing its value with the percentage measured. As for the ratio of distinct values, the result collection is iterated and, record by record, an auxiliary data structure containing key pairs of <*ColumnName*, *HashTable*<*Values*>> is updated, by adding values to the hash table (if they do not yet exist

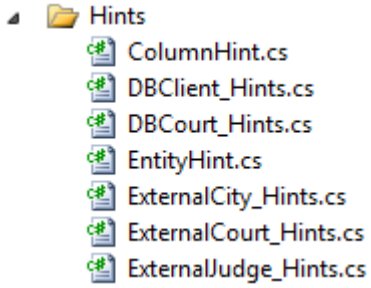


Figure A.28: Package containing the implementation of hints

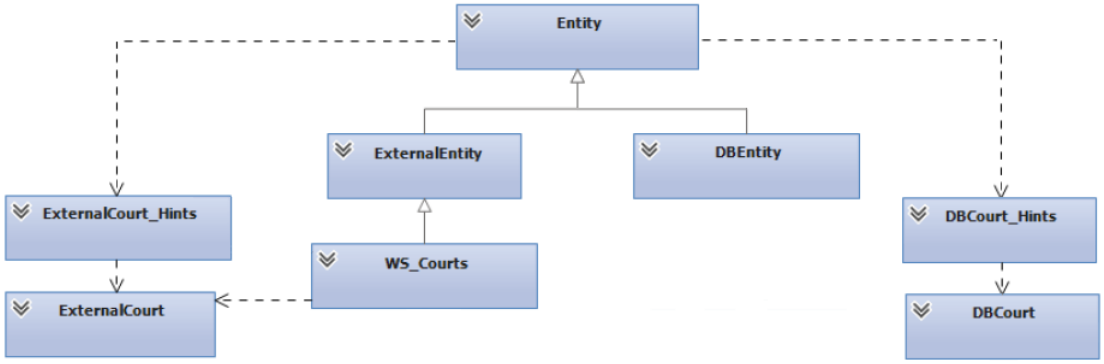


Figure A.29: General model structure

there). In the end, the number of distincts for each column is the size of its hash table and the percentage is computed and stored in the entity collection *distinctRatios*, by updating the dictionary in the related key (column name), replacing its value with the percentage measured. Once again, we speak only of APIs since these measures for database entities are available in a catalog.

Both of these distribution summaries (distincts and nulls) are created/updated in the same iteration loop, that is, by iterating a result collection once the system is capable of building both structures. Thus, no extra overhead is added to the algorithm by these features.

A.3.3.2 Hints implementation

The classes used for the annotations (*ColumnHint*, *EntityHint*) and the classes to retrieve those hints from each entity (*ExternalCourt_Hints*, for instance) are stored inside the package *Hints*, as shown in Figure A.28. Inside the class *Entity*, whenever a statistic is consulted and it is not available, the related hint is thereby checked. Thus, *Entity* class knows which entity class should be inspected and invokes the specific class that searches the hint (*ExternalCourt_Hints*, for example).

The general model for the statistics and hints data model is presented in Figure A.29, with two entities inserted on it: a database entity *DBCourt* and a web service *WS_Courts*.